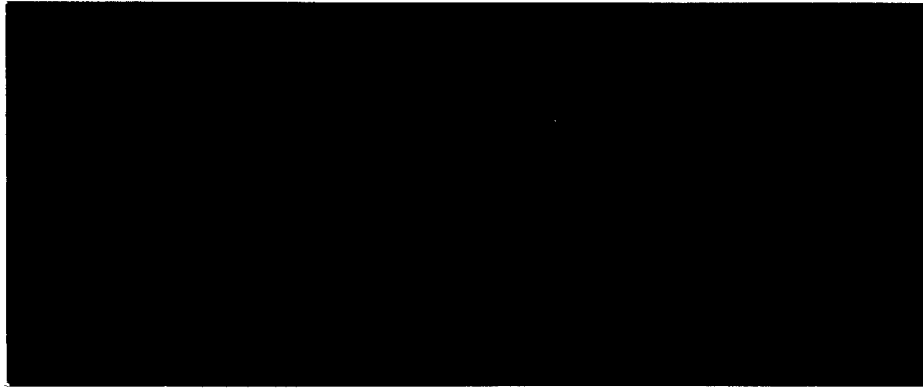

Computer Science



19971007 146

EXTRA QUALITY INSPECTED 4

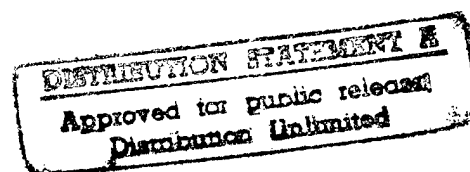
**Carnegie
Mellon**

DISTRIBUTION STATEMENT A
Approved for public release
Distribution Unlimited

Operating System Resource Reservation for Real-Time and Multimedia Applications

Clifford W. Mercer

June 1997
CMU-CS-97-155



School of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213-3890

*Submitted in partial fulfillment of the requirements for
a degree of Doctor of Philosophy in Computer Science*

Thesis Committee:

Elmootazbellah N. Elnozahy, Co-chair
Ragunathan Rajkumar, Co-chair
Mahadev Satyanarayanan
Hideyuki Tokuda
Kevin Jeffay, University of North Carolina-Chapel Hill

Copyright ©1997 Clifford W. Mercer

This research was supported in part by a National Science Foundation Graduate Fellowship, in part by Bell Communications Research, in part by U. S. Naval Research and Development, in part by the Office of Naval Research, in part by Northrop-Grumman, in part by Philips Research, and in part by Loral Federal Systems. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of NSF, Bell Communications Research, NRaD, ONR, Northrop-Grumman, Philips Research, or Loral Federal Systems.

Keywords: Resource reservation, Real-time systems, Operating systems, Multimedia applications, Scheduling, Resource management, Real-Time Mach



School of Computer Science


DOCTORAL THESIS
in the field of
Computer Science

*Operating System Resource Reservation
for Real-Time and Multimedia Applications*

CLIFFORD MERCER

Submitted in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy

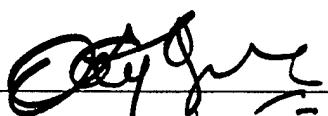
ACCEPTED:



THESIS COMMITTEE CHAIR

6-24-97

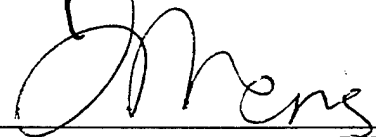
DATE



THESIS COMMITTEE CHAIR

6/30/97

DATE

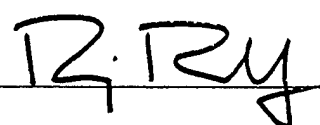


DEPARTMENT HEAD

8/7/97

DATE

APPROVED:



DEAN

8/10/97

DATE

For Franke

Abstract

Increases in processor speeds and the availability of audio and video devices for personal computers have encouraged the development of interactive multimedia applications for teleconferencing and digital audio/video presentation among others. These applications have stringent timing constraints, and traditional operating systems are not well suited to satisfying such constraints. On the other hand, hard real-time systems that can meet these constraints are typically static and inflexible.

This dissertation presents an enforced operating system resource reservation model for the design and implementation of predictable real-time programs. Applications can reserve resources based on their timing constraints, and an enforcement mechanism ensures that they do not overrun their reservations. Thus, reserves isolate real-time applications from the temporal properties of other real-time (and non-real-time) applications just as virtual memory systems isolate applications from memory accesses by other applications. In addition, reserves are first class objects that are separated from control abstractions such as processes or threads. Therefore reserves can be passed between applications, and this model extends naturally to distributed systems.

Reserves support the development of hard real-time and soft real-time programs, and programming techniques based on reserves illustrate how to use them effectively. An implementation of processor reserves in Real-Time Mach shows that reserved multimedia applications can achieve predictable real-time performance.

Acknowledgments

I would like to express my appreciation to my advisors, Mootaz Elnozahy and Raj Rajkumar, for their guidance, technical expertise, and time which they so generously shared with me during my doctoral studies. Thanks also to M. Satyanarayanan, Hide Tokuda, and Kevin Jeffay for their insights, critical comments, and encouragement. I am especially indebted to Hide Tokuda who was my mentor for several years before I entered the graduate program at CMU. Hide started the Real-Time Mach Project, which provided the research environment for this work, and Raj took over the project in the final years of my thesis work. I am indebted to both of them for their support.

I would like to thank Jim Morris for his guidance and assistance. He helped me better understand how to do systems research and served as a role model for me. Thanks go to Roger Dannenberg for the opportunity to work with him and learn from him on several interesting computer music projects (and for the occasional trumpet duet and brass ensemble session).

A special thanks to Joan Maddamma who looked out for me in many ways during my years at CMU. From the time I was an undergraduate through my years as a graduate student, Joan has been a very special friend.

Many other people who I worked with over the years had a positive influence on me and on the work contained in this dissertation. I would like to thank Stefan Savage and Jim Zelenka for their help with pieces of the implementation of reserves in Real-Time Mach and for being good friends and colleagues. And I would like to thank Chen Lee, Prithvi Rao, Andrei Ghetie, Yutaka Ishikawa, Tatsuo Nakajima, Takuro Kitayama, Dan Katcher, Kevin Kettler, John Sasinowski, and Saurav Chatterjee for being part of the invigorating academic environment that makes CMU so rewarding.

I would like to thank my father, mother, and sister for their moral support during my years away at school.

Finally, I want to thank my wife, Franke, who supported me wholeheartedly in the pursuit of my graduate studies. Without her, this work would not have been possible. Thanks also to my daughters, Caroline and Leigh, for their love and patience with me during the final years of the thesis work.

Table of Contents

1	Introduction.....	1
1.1	Motivation	1
1.2	Background.....	2
1.2.1	Programming real-time applications	2
1.2.2	Resource management problems	3
1.3	Resource reserves	5
1.4	Contributions	7
1.5	Overview of the dissertation.....	7
2	Background and Motivation	9
2.1	Real-time and multimedia application requirements	9
2.1.1	Timing characteristics	9
2.1.2	Criticality	13
2.2	Applications timing requirements	16
2.3	Quality of service management	18
2.3.1	QOS background	18
2.3.2	Mapping QOS parameters	19
2.3.3	QOS negotiation.....	20
2.4	System design approaches	21
2.4.1	Specialized hardware	21
2.4.2	Time-sharing systems.....	21
2.4.3	Real-time operating systems	22
2.4.4	Soft real-time system support.....	23
2.5	Reserve abstraction.....	24

3	Reserve Model	25
3.1	Reserve abstraction	25
3.1.1	Reservation guarantee	25
3.1.2	Scheduling frameworks	26
3.1.3	Styles of programming with reserves	26
3.2	Basic reserves	27
3.2.1	Reservation specification	28
3.2.2	Admission Control	29
3.2.3	Scheduling	29
3.2.4	Enforcement	30
3.3	Reserve propagation	33
3.4	Example scheduling frameworks	34
3.4.1	Rate monotonic	35
3.4.2	Deadline monotonic	36
3.5	Basic reserve types	38
3.5.1	Processor	39
3.5.2	Physical memory	40
3.5.3	Network bandwidth	41
3.6	Reserve management	42
3.6.1	Default reserves	42
3.6.2	Composite reserves	42
3.6.3	Reserve inheritance	43
3.7	Chapter summary	44
4	Programming with Reserves	45
4.1	Overview	45
4.2	Using reserves in application design	46
4.2.1	Program structure	46
4.2.2	Reservations for periodic computations	48
4.2.3	Localized reserve allocation	49
4.2.4	Activity-based reserve allocation	52
4.2.5	Coordinating multiple resources	53
4.3	Sizing Reservations	58
4.3.1	Determining resources required	59
4.3.2	Determining initial reservation levels	60
4.3.3	Measuring performance	61
4.3.4	Adapting	67
4.4	Chapter summary	71

5	Implementation	73
5.1	Overview	73
5.2	Reserves in RT-Mach	74
5.2.1	Attributes and basic operations	74
5.2.2	Reservation requests and admission control	76
5.2.3	Scheduling	77
5.2.4	Usage measurement and enforcement	78
5.2.5	Reserve propagation	79
5.3	Applications	81
5.3.1	QuickTime video player	81
5.3.2	MPEG decoder	82
5.3.3	X Server	83
5.4	Reserved network protocol processing	85
5.4.1	Software interrupt vs. preemptive threads	85
5.4.2	Mach 3.0 networking	86
5.4.3	Reserved protocol processing	88
5.5	QOS manager	88
5.5.1	Information sources	88
5.5.2	Admission control	90
5.5.3	Extensions	90
5.6	Tools	91
5.6.1	Reserve monitor	91
5.6.2	Usage monitor	93
5.7	Chapter summary	94
6	Experimental Evaluation	97
6.1	Overview	97
6.2	Predictability	99
6.2.1	Independent synthetic workloads	101
6.2.2	Client/server synthetic workloads	107
6.2.3	QTPlay/X Server	115
6.2.4	mpeg_play/X Server	122
6.2.5	Protocol processing workloads	124
6.3	Scheduling cost	131
6.3.1	Measured aggregate scheduling cost	132
6.3.2	Individual operations	133
6.4	Chapter summary	136

7 Related Work	137
7.1 System Implementation	137
7.1.1 Multimedia support	137
7.1.2 QOS architectures	139
7.1.3 Networking	140
7.2 Scheduling theory	141
7.3 Applications	142
7.3.1 Adaptive applications	142
7.3.2 Tools	142
8 Conclusion	145
8.1 Contributions	145
8.1.1 Resource reservation abstraction	145
8.1.2 Implementation	146
8.1.3 Experimental evaluation	146
8.2 Future directions	147
Bibliography	149

List of Figures

2-1	Schematic of a Time Constrained Computation	10
2-2	Periodic Task	11
2-3	Aperiodic, Predictable Task	11
2-4	Aperiodic, Unpredictable Task	12
2-5	Preemptible Task	12
2-6	Non-preemptible Task	13
2-7	Hard (Catastrophic) Deadline Value Function	13
2-8	Hard Deadline Value Function	14
2-9	Ramped Hard Deadline Value Function	14
2-10	Soft Deadline Value Function	15
2-11	Non-real-time Value Function	15
2-12	Periodic Playback Computations	16
2-13	Playback Application Computing Ahead	17
2-14	Interactive Application with Limited Workahead	17
2-15	Levels of QOS Specification	19
2-16	QOS Levels with QOS Manager	20
3-1	Periodic Scheduling Framework	29
3-2	Enforcement Illustration	30
3-3	Enforcement Timers	32
3-4	Reserve Propagation	33
3-5	Deadline Monotonic Scheduling Framework	37
3-6	Resources and Basic Reserves	38
3-7	A Composite Reserve	43
3-8	Reserve Inheritance	44
4-1	Call Graph for Frame Generation and Display	47
4-2	Thread and Reserve Out-of-Phase	48
4-3	Call Graph with Separate Client and Server Reserves	49
4-4	Switching Reserve from Client to Server	50
4-5	Client Requirement with Intermediate Deadline	51
4-6	Call Graph with One Reserve for All Nodes	52

4-7	Call Graph for Video Playback	54
4-8	Call Graph for Video Playback with Reserves	55
4-9	Synchronization Problem with Multiple Resources	56
4-10	Multiple Resources Used with Intermediate Deadlines	57
4-11	Video Playback with Better Reserve/Computation Mapping	58
4-12	Resource Demand Constant and Reserved	62
4-13	Resource Demand Occasionally Exceeds Reservation	63
4-14	Exceedingly Demanding Computation Aborted	64
4-15	Computation Impinges on Following Computation	64
4-16	Computation Impinges on Subsequent Computations	65
4-17	Average Demand Exceeds Reservation	66
4-18	Resource Demand Smaller than Reservation	66
4-19	Measurements of Multiple Resources	67
5-1	System Components	73
5-2	QTPlay Outline	81
5-3	mpeg_play Outline	82
5-4	Networking with the UX Server	86
5-5	Networking with the Socket Library	87
5-6	Resource Management Schematic	89
5-7	rmon Main View	91
5-8	rmon Detail Views	92
5-9	Modifying a Reservation	94
6-1	Compute-Bound Periodic Task with No Competition	99
6-2	Compute-Bound Periodic Task with Competition	100
6-3	Experiment 1 Results	104
6-4	Experiment 2 Results	105
6-5	Experiment 3 Results	106
6-6	Experiment 4 Results	110
6-7	Experiment 5 Results	111
6-8	Experiment 6 Results	112
6-9	Experiment 7 Results	113
6-10	Experiment 8 Results	114
6-11	Software Configuration	116
6-12	Experiment 9 Results	118
6-13	Experiment 10 Results	119
6-14	Experiment 11 Results	120
6-15	Experiment 12 Results	120
6-16	Experiment 13 Results	121
6-17	Experiment 14 Results	123
6-18	Experiment 15 Results	127
6-19	Experiment 16 Results	129
6-20	Experiment 17 Results	130
6-21	Experiment 18 Results	131
6-22	Scheduling Cost	133

List of Tables

6-1	Summary of Testbed Platforms	98
6-2	Experiment 1 Parameters	102
6-3	Example Parameters for Experiment 2	102
6-4	Example Parameters for Experiment 3	103
6-5	Experiment 4 Parameters	108
6-6	Experiment 6 Parameters	108
6-7	Experiment 7 Parameters	109
6-8	Experiment 8 Parameters	109
6-9	Experiment 9 Parameters	116
6-10	Experiment 10 Parameters	117
6-11	Experiment 11 Parameters	117
6-12	Experiment 12 Parameters	117
6-13	Experiment 13 Parameters	118
6-14	Experiment 15 Parameters	125
6-15	Experiment 16 Parameters	125
6-16	Experiment 17 Parameters	126
6-17	Experiment 18 Parameters	126
6-18	Reserve Switch	134
6-19	Replenishment Timer	135
6-20	Checkpoint Cost	135

Chapter 1

Introduction

This dissertation presents the design, implementation, and experimental analysis of a model for operating system resource reservation. The reservation system supports predictable performance in real-time and multimedia applications, enabling them to meet their timing requirements, and facilitating adaptive resource management. This approach is suitable for real-time programming problems that arise in personal computers and workstations where users may want to run real-time multimedia applications or other real-time programs. The approach is also applicable to embedded system design where better resource reservation abstractions at the system level aid in the design, debugging, and maintenance of such systems.

1.1 Motivation

Recent increases in processor speed and network bandwidth combined with the wide availability of digital audio and video devices have enabled a plethora of multimedia applications and services. Examples of these include audio/video presentation and playback, audio/video phone and conferencing, persistent multimedia data storage services, telephone answering and call management, speech processing, and virtual reality applications.

Stringent timing constraints and large volumes of data characterize these applications. Existing operating systems are not designed to support such applications, especially when real-time multimedia applications execute alongside a non-real-time workload. A key requirement of systems for multimedia applications is predictability, and this means that possible contention for resources must be identified and managed to ensure the timeliness of multimedia data processing and delivery. Although it is possible to manage contention for system resources in an *ad hoc* manner based on the specific requirements of a particular class of applications, this dissertation describes a more structured approach to managing contention based on:

- A reservation model that provides an abstraction for resource capacity

reservation and a system-level mechanism for scheduling resources.

- A higher-level layer for implementing resource management policy using the system-level mechanism.
- Programming techniques for structuring applications in a way that can take full advantage of the resource reservation model.

The reservation mechanism and allocation policy provide abstractions that relieve the system designer from relying on complicated high-level application information to make low-level scheduling decisions. And the programming techniques facilitate the programming of real-time applications that meet their timing constraints.

1.2 Background

Real-time system designers must take timing constraints into account when developing real-time applications and the systems to support them. The programming techniques and resource management policies that have been developed for real-time systems typically apply *ad hoc* solutions for each application area. Several issues and problems arise which can be addressed with appropriate system abstractions.

1.2.1 Programming real-time applications

In applications with timing requirements, the software must be designed to satisfy the timing constraints. The user-level servers and system services used by such applications must be designed with timing constraints in mind. The resource management policies underlying those system services must also be designed to support applications with timing constraints.

Traditionally in real-time system design, application programmers use small, simple operating systems that provide fixed priority processor scheduling, priority queueing for various system resources such as semaphores and mailboxes, and perhaps priority inheritance protocols. The application programmers must then build many of their own real-time system services such as database management systems, file systems, and network communication. These programmers must carefully schedule the various applications running on the system and manage contention for the processor and other system resources. In doing the design and scheduling, the computational requirements of applications must be carefully measured and characterized, and resource sharing must be carefully planned. Applications are therefore very sensitive to the misbehavior of other applications. For example, if a high-priority real-time application has a bug that sends it into an infinite loop, the effect on other applications and the system as a whole would be devastating. The errant application would take over the processor and would not relinquish it, forcing a system crash.

This extreme sensitivity among applications is the result of a lack of suitable system abstractions for effectively managing shared resources in real-time systems. Many abstractions exist in real-time scheduling theories, but typically the assumptions of the theoretical results are implicitly embedded in real systems. An example of an assumption that many

theories require is that the worst-case execution times (WCET) for computations in real-time applications are known at system design time. Most systems designed using analysis techniques that have this assumption do not explicitly check or enforce worst-case execution times for computations. Appropriate system abstractions would explicitly bring the assumptions into the actual system where they could be checked and, in the best case, even enforced.

1.2.2 Resource management problems

Suitable system abstractions could effectively address several problems that arise in real-time system. These problems fall into three broad areas: resource management policies that can satisfy timing constraints, mechanisms to support the policies, and analysis techniques based on the available mechanisms.

Many systems do not provide scheduling policies that directly support real-time resource management. For processor scheduling, most systems provide either fixed priority or deadline scheduling policies. Both of these policies lack complete information about real-time requirements and therefore do not address important problems such as how priority should be assigned or how to prevent overload. The following issues arise in the design of resource management policies:

- **Priority assignment problem:** Simple priority schedulers are hard to use, especially if there are many activities with timing constraints. If there is no global repository of knowledge about the timing constraints of different activities in different applications, there is no basis for deciding what the priority ordering of the activities should be.
- **Overload problem:** To prevent overload, the scheduling policy requires information about real-time constraints such as the computational requirements and frequency of execution. Even if the designer can express the resource and timing requirements for real-time applications, a system with no admission control policy cannot protect itself from overload.
- **Flexibility requirement:** The timing constraints and resource requirements for dynamic real-time applications may change dynamically during execution. The scheduling algorithm must support efficient adjustment of requirements.

To effectively schedule real-time applications such that applications cannot monopolize system resources requires some usage measurement and enforcement mechanisms. Information gleaned through these mechanisms can be used in the scheduling policy to make decisions about how priorities or deadlines should be dynamically adjusted to reflect the requirements and usage patterns of applications. The problems that motivate the use of these mechanisms are described briefly below:

- **Enforcement problem:** An application that specifies resource and timing requirements may accidentally or deliberately attempt to exceed its

stated requirements. This may interfere with the satisfaction of timing constraints for other reserved activities, and it may cause starvation among unreserved activities.

- **Measurement problem:** Enforcement of resource requirements means that resource usage of each application must be accurately measured and compared to the allocation that has been made on its behalf. If the activity uses external modules, servers, and system services, the measurement must include that usage as well.
- **Coordination problem:** Many time-constrained activities are composed of multiple sub-activities implemented by other modules, user-level servers, or system services. Allocating resources for a single activity that spans multiple modules, possibly in different memory protection domains, is complicated. It is necessary, however, to be able to place timing requirements on the overall activity and to track the resource usage for the overall activity.

Other problems deal with higher level issues of how to analyze system behavior given more sophisticated abstractions and mechanisms for scheduling and resource management. The following issues arise in this context:

- **Distributed real-time scheduling problem:** An activity that uses external modules and services may require the use of multiple resources within certain time constraints. Coordinating usage across multiple resources to meet timing constraints is a hard problem.
- **QOS management problem:** Real-time applications may dynamically change their quality of service (QOS) requirements. In a system with many such applications, a high-level resource manager (sometimes called a QOS manager) is needed to resolve conflicts and negotiate between applications.

In traditional real-time systems design, many of these problems are avoided in the design phase by careful measurement and analysis of simple computations and their resource requirements and by ad hoc scheduling techniques. This approach results in inflexible systems that are difficult to maintain [43,68]. In particular, supporting dynamic real-time activities with timing constraints and resource requirements that may change freely at run-time stretches the traditional approach beyond its limits.

Recent work in real-time systems addresses some of these problems. Several systems (e.g. [3,53,113,124,125]) allow specification of timing requirements instead of forcing the programmer to determine an appropriate priority assignment. A few systems have on-line admission control policies [3,78,113], but many others use off-line analysis [53,124,125]. Still others have no admission control at all [19,21,90]. Some limited flexibility requirements for hard real-time have been addressed recently [122]. This work focuses on mode changes, which are radical but infrequent changes in the task sets of a real-time system. For example, the real-time system in an airplane might experience a mode change after takeoff as it switches from the ground-based task set to the air-based task set.

Very few software systems address the measurement and enforcement problems with respect to resource usage although most systems can detect and react to missed deadlines [43,125]. In networks, the notion of enforcement or policing is much more common [33,98,128]. The work on priority inheritance protocols [8,16,51,86,97,108] addresses some aspects of the coordination problem.

The distributed real-time scheduling problem is an active area of research [38]. The QOS problem is another active area of research. Some operating system research in this area focuses on best effort approaches [21] although other research emphasizes guarantees [46,53,78,81,126].

1.3 Resource reserves

This dissertation defines a resource reservation model that provides an operating system abstraction called a reserve. Reserves explicitly represent reservations on resources such as processors, memory pages, disks, and network devices. In particular, reserves support

- specification of reservation parameters,
- admission control,
- scheduling based on timing constraints and usage requirements,
- reservation enforcement,
- reserve propagation in the RPC mechanism,
- flexible binding of threads to reserves.

Reserves prevent applications from over-running their allowed resource usage and interfering with other reserved activities or starving unreserved activities. Applications reserve capacity on the resources they need to carry out their computations. For example, an application can reserve 10 ms of computation time on a processor for every 100 ms of real-time. The application then binds to the reserve, and the processor scheduler uses the information associated with the reserve to control the scheduling of the application. The system also performs an admission control test before granting the reservation to make sure that the resource can support the reservation being requested. The enforcement mechanism ensures that an application does not use more than its reserved time if doing so would interfere with other reserved activities.

The reservation parameters associated with an application's reserves are not necessarily fixed for the lifetime of the application. A dynamic real-time application must be prepared to change its behavior and timing requirements based on changing requirements of users and possibly the changing availability of resources. A user may want to change the frame rate on a video player or change the resolution, and the application must be ready to adjust its resource reservation levels appropriately. Likewise, reserves must support an operation that modifies the reservation parameters, subject to the admission control policy.

Reserves are first class objects in the operating system; a reserve is associated with a particular thread (or process) by explicitly binding the thread to the reserve. This allows an

application to reserve all of the resources it will need for its computation, including resources that will be needed by various modules, servers, and system services it intends to invoke. The application can then pass references for its resource reserves to modules or servers along with the operation invocation. The module or server can bind its thread to this reserve when performing the operation, and it can then take advantage of the resources that have been reserved by the client. Having modules and servers charge a caller's reserve for work done on behalf of the caller also maintains a consistent view of the resource usage that is being consumed on behalf of that client.

The reserve model presented in this dissertation addresses the problems identified in the previous section. The scheduling policies embedded in the reserve model address the priority assignment problem and the overload problem while remaining flexible in terms of accommodating dynamic changes in application resource requirements as discussed below.

- **Priority assignment problem:** Reserves avoid the priority assignment problem by accepting reservation specifications that include the timing constraints and usage requirements.
- **Overload problem:** The admission control policy of the reservation mechanism prevents overload. This is possible since the scheduling policy has information about both the computational resource requirements and the timing constraints (such as period of invocation) for each application.
- **Flexibility requirement:** Changes in reservation parameters can be made at any time, subject to the admission control policy.

The reserve model makes use of several system mechanisms that support the abstraction. These mechanisms provide information about resource usage and that coordinate the consumption of resources for an activity that crosses memory protection boundaries as follows:

- **Enforcement problem:** Reservation enforcement isolates reserved activities from undue interference from other reserved activities.
- **Measurement problem:** The flexibility in binding reserves makes it possible to accumulate resource usage charges for an activity even when work is done by external modules, servers, or system services.
- **Coordination problem:** The reserve model supports reserve propagation which includes a "priority" inheritance mechanism and which takes advantage of the flexibility in binding reserves to threads.

The reserve model provides an abstraction that can be used for distributed real-time scheduling and QOS management. The approaches to these problems are described briefly below:

- **Distributed real-time scheduling problem:** The reserve model supports reservations for remote resources, and pipeline-style software architectures are supported. This is not optimal, but future work on this problem could take advantage of the reserve model as a base.

- **QOS management problem:** A high-level QOS management layer can use reserves to carry out resource allocation policy decisions. The QOS managers can carry out their allocation decisions by manipulating reservation parameters for the applications being managed.

1.4 Contributions

This dissertation describes and analyzes a resource reservation solution to the problem of supporting predictable execution of real-time and multimedia applications with specific quality of service parameters. It shows that:

Resource reserves, an enforced operating system resource reservation abstraction, effectively supports real-time and multimedia applications. Reserves allow the software designer to specify timing requirements on resources required, thus providing a method for guaranteeing deadlines in real-time applications. The reservation abstraction accommodates non-real-time applications as well as real-time applications.

This dissertation defines an enforced resource reservation model called reserves and then describes programming techniques for developing applications using reserves. It presents an implementation of one type of resource reserves, processor reserves, in Real-Time Mach along with several real-time applications that use reserves to satisfy their timing constraints. These applications included a suite of synthetic benchmark programs, a QuickTime video player, an MPEG player, and a version of the X server. Experiments with these applications showed that reserves can indeed provide predictable behavior for real-time applications even with competition from other real-time and non-real-time applications.

Reserves are a tool that real-time system designers can use to raise the level of abstraction for resource scheduling in real systems. This allows applications' timing requirements to be defined and guaranteed in isolation with the system providing high-level guarantees that resources will be available when needed by each real-time application.

1.5 Overview of the dissertation

Chapter 2 describes the background and motivation for the dissertation in more detail. Chapter 3 describes the reserve model, and Chapter 4 discusses techniques for structuring programs to best take advantage of reserves.

Chapter 5 describes the implementation of processor reserves in Real-Time Mach, a quality of service (QOS) manager, and several reserved applications and servers. Chapter 6 presents an experimental evaluation that explores the predictability achieved by using reserves and the overhead involved in providing that predictability.

Chapter 7 discusses related work, and Chapter 8 summarizes the contributions of the dissertation and presents the conclusions and future directions.

Chapter 2

Background and Motivation

This chapter discusses the requirements of real-time and multimedia applications and describes some of the problems system designers encounter in attempting to support such applications. The case is made for an operating system resource reservation approach to the problem.

2.1 Real-time and multimedia application requirements

Real-time applications require not only that computations result in logically correct answers, but that the answers are available within certain timing constraints. A logically correct answer that arrives late is considered incorrect in a real-time system [114]. Many multimedia applications have this property that late computations are useless. For example, if a video frame or audio sample arrives after the time at which it was to be displayed or played, it is no longer useful.

This section gives an overview of different kinds of timing constraints and criticality characteristics of real-time and multimedia applications. The models described here represent a compendium of the models that researchers have addressed in the literature.

2.1.1 Timing characteristics

In general, a task in a real-time system has timing constraints that specify when the computation may begin, how long it executes, and the deadline for the completion. Figure 2-1 illustrates a computation schematically. This generic model is common in the operations research literature [20]. The computation has a ready time, r , at which the computation becomes available for scheduling. At some point after the ready time, the computation will be scheduled and start processing for a total duration of C . The duration between the ready time and the start of processing is enclosed in a white box. This indicates that the task is available for scheduling but has not started yet. The black box represents the computation that completes at time E . A deadline, d , may be associated with the computation as well, and

the goal is to complete the computation before the deadline. In Figure 2-1, a thick vertical line represents the deadline.

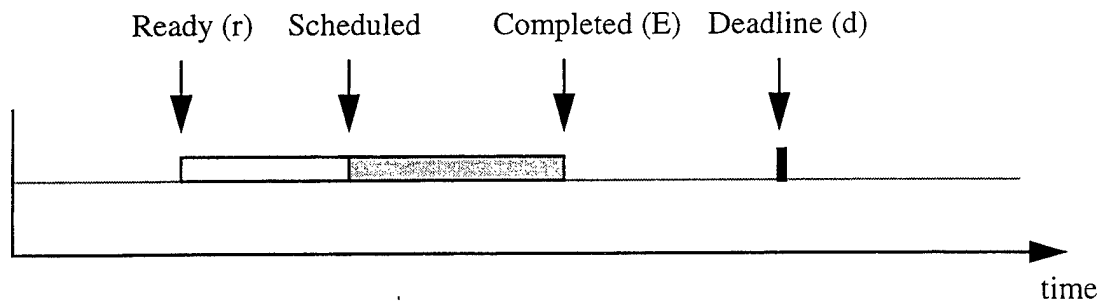


Figure 2-1: Schematic of a Time Constrained Computation

The ready time of a computation may arise from a clock event, an external interrupt, or a software event generated by some other computation. The ready event may be an instance of a periodic computation where the same computation is activated periodically. The ready event may be aperiodic but predictable, or it may be unpredictable. The computation time may be fixed or it may be variable or unpredictable. The computation itself may be preemptible, or it may form a non-preemptible critical region. The deadline is usually some fixed duration after the ready time, but the implications of missing a deadline may vary. Hard real-time computations take the deadline to be a *hard* deadline where the computation must be complete by the deadline time. Alternatively, the deadline may just be a recommendation or preference for completion of the computation, a *soft* deadline.

Since a computation may be periodic, we must sometimes distinguish between the overall activity and the periodically occurring computations. We call the overall activity a *task*, and we refer to an instantiation or individually scheduled computation of the task as a *job*; thus a task is a stream of jobs. The jobs of a particular task are considered to be identical for the purposes of scheduling although variations can be indicated by a variable or stochastic computation time. We will use the word task to mean both the stream of instantiations and an individual instantiation when such usage is clear from the context.

A periodic task has ready times for the task instantiations separated by a fixed period. Periodic tasks are the main focus of the original rate monotonic scheduling work [67] and the many extensions that have followed [63,108]. Figure 2-2 shows a periodic task. The figure shows four instantiations, each with an associated ready time, r_i . The ready times are separated by the period τ . The computation time is represented by the black box, and the preceding white box represents the time when the task is ready by has not yet been scheduled to execute. In this example, the computation time is constant across task instantiations, and the deadline is at the end of the period.

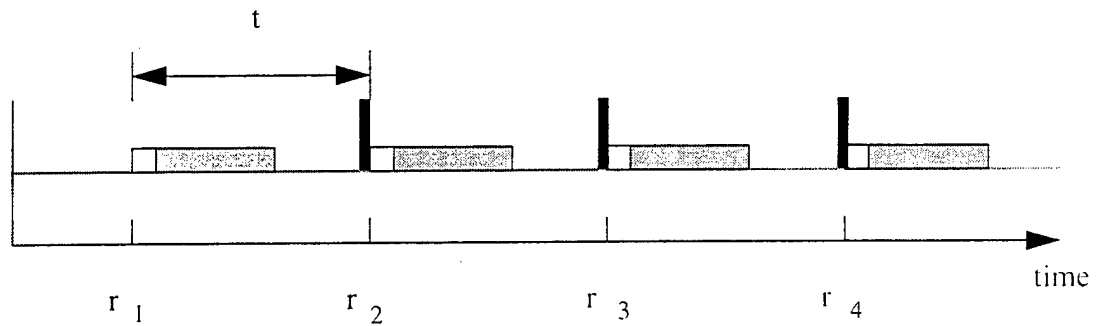


Figure 2-2: Periodic Task

Aperiodic tasks are more difficult to specify. Some aperiodic tasks are predictable to a certain extent; it may be possible to predict the arrival of instantiations of an aperiodic task within some scheduling horizon of h time units. Figure 2-3 shows an aperiodic task with a scheduling horizon of duration h from the current time. This kind of timing requirement is used in computer music, for example [5,25]. Within this window of h time units, the ready times of instantiations of the task are known, but beyond the horizon, nothing is known of the behavior. The computation time is assumed to be constant across instantiations in the single task, and the deadlines are left unspecified.

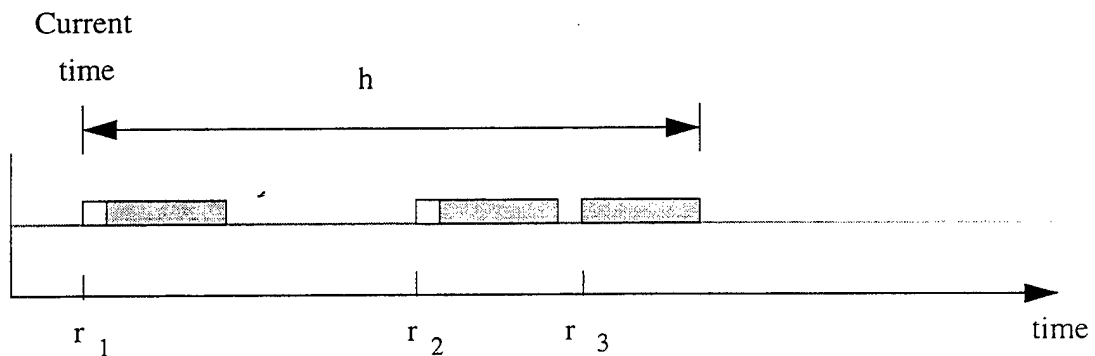


Figure 2-3: Aperiodic, Predictable Task

Another class of aperiodic tasks is almost completely unpredictable. It is common, however, to associate a minimum interarrival time for the instantiations of these unpredictable aperiodic tasks. Much work has been done on scheduling aperiodic tasks with soft deadlines [120] and on aperiodic tasks with hard deadlines, which are known as *sporadic* tasks [52,111]. Figure 2-4 illustrates an aperiodic task where the arrivals are unpredictable.

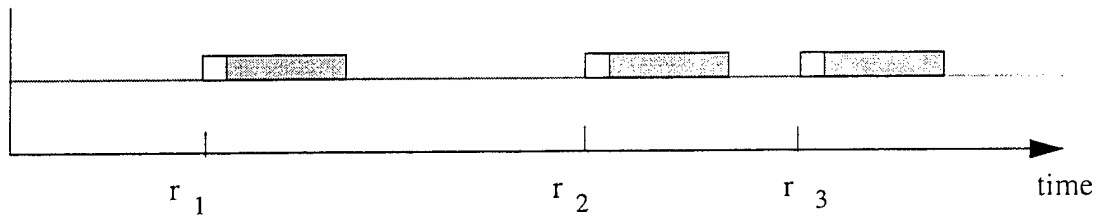


Figure 2-4: Aperiodic, Unpredictable Task

The computation time is another dimension along which tasks may vary. The computation time may be fixed or merely bounded in duration. The computation could also be described by a statistical distribution, but that case is much harder to analyze.

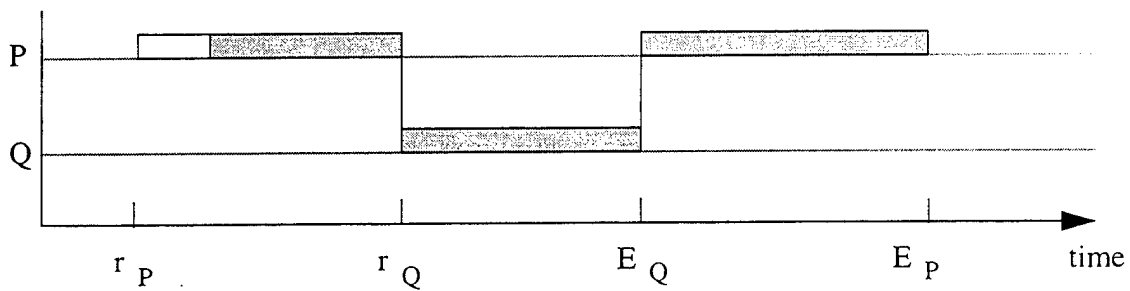


Figure 2-5: Preemptible Task

Another characteristic of the computation is its preemptibility. It may be completely preemptible (that is preemptible at any point) or it may be non-preemptible. Or it may be preemptible but with one or more non-preemptible critical regions during which scheduling events are not allowed (possibly during system calls for example). Different assumptions are made to achieve different results [67,72,82], and in particular, much work has been done on handling non-preemptible critical regions [8,16,51,97,108]. Figure 2-5 shows an example of a preemptible task, P, and its interaction with another task, Q. For this example, we assume that P is preemptible and has a lower priority than Q. P becomes ready at time r_P and begins to execute immediately. At time r_Q , Q becomes ready, and since Q has priority over P, Q preempts the ongoing execution of P. After Q completes, the execution of P resumes.

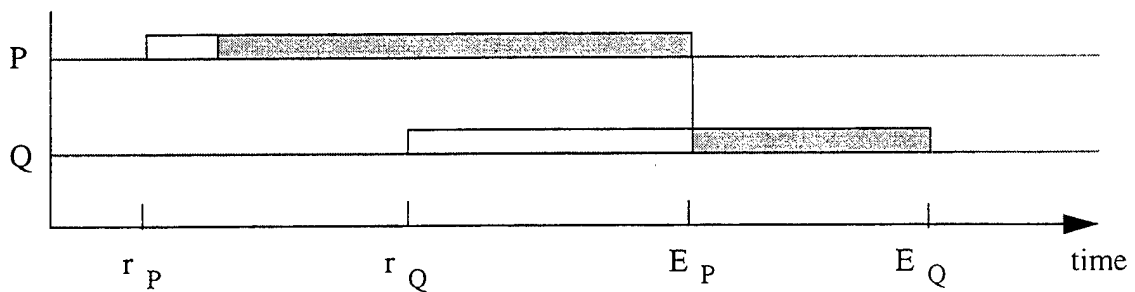


Figure 2-6: Non-preemptible Task

Figure 2-6 illustrates a similar case where the computation of P is non-preemptible and where Q has priority over P. P becomes ready at time r_P and begins to execute. Q becomes ready at time r_Q , but even though Q has priority over P, P cannot be preempted, and Q must wait until the execution of P completes. After P is finished, Q can begin execution.

2.1.2 Criticality

Deadlines may be classified as hard or soft. We can describe various types of deadlines by means of a value function. Value functions have been used for scheduling [15,55], but here they are used for purposes of exposition. A value function is a function of time that indicates the value that completion of the task would contribute to the overall value of the system. For example, Figure 2-7 shows the value function of a task that has a hard deadline: the value drops off to negative infinity at $t = d$. The task becomes ready at time r , and its deadline is d . If the task is completed at time t where $r \leq t \leq d$, then the system receives some value, V . On the other hand, if the task completes after d , the value is negative infinity, a catastrophic failure.

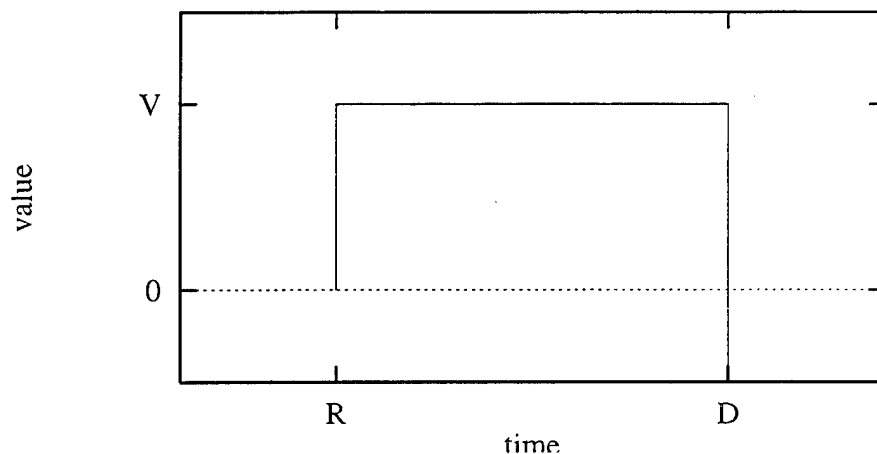


Figure 2-7: Hard (Catastrophic) Deadline Value Function

The result of missing a hard deadline may not be catastrophic. Figure 2-8 shows a case where completion of a task would have some value until the deadline d when the value of completion of the task goes to zero. This indicates that the system will receive no benefit from completing the computation after d , and so the task should be aborted, freeing any resources it holds. In contrast to the previous case, the system can continue to operate, achieving a positive value even if this particular task is aborted and makes no contribution to that value.

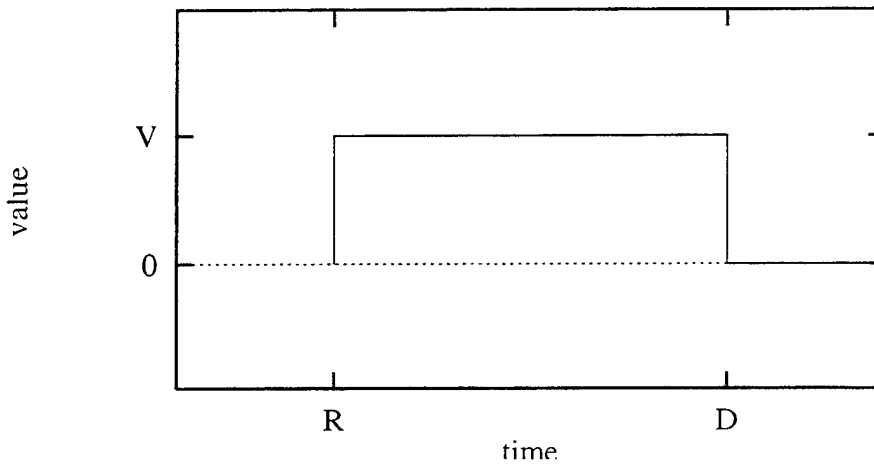


Figure 2-8: Hard Deadline Value Function

Other variations on the idea of hard deadline might include a value function that ramps up to the deadline as illustrated in Figure 2-9. And depending on where the ramp starts, this type of value function can specify tasks that must be executed within very narrow intervals of time.

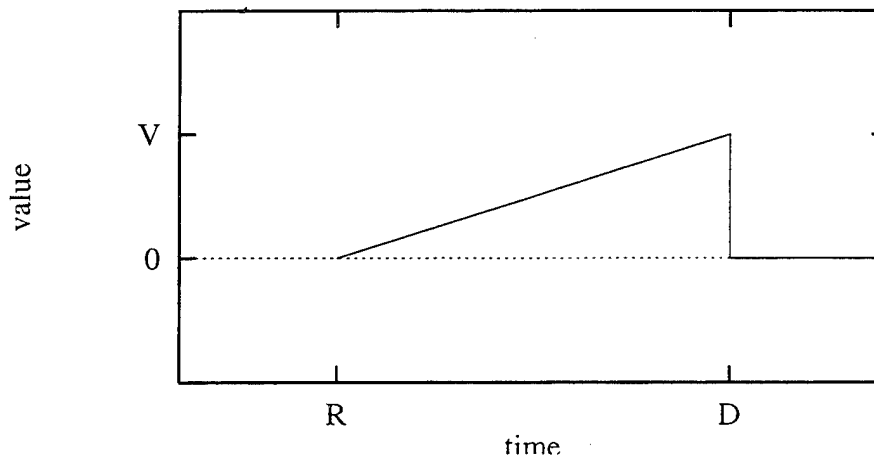


Figure 2-9: Ramped Hard Deadline Value Function

The concept of a soft deadline is illustrated in Figure 2-10 where the value function goes gradually to zero after the deadline. In this case, there is some value to the system in completing the task after the deadline, and the task should not be aborted right away as in the case of the hard deadline.

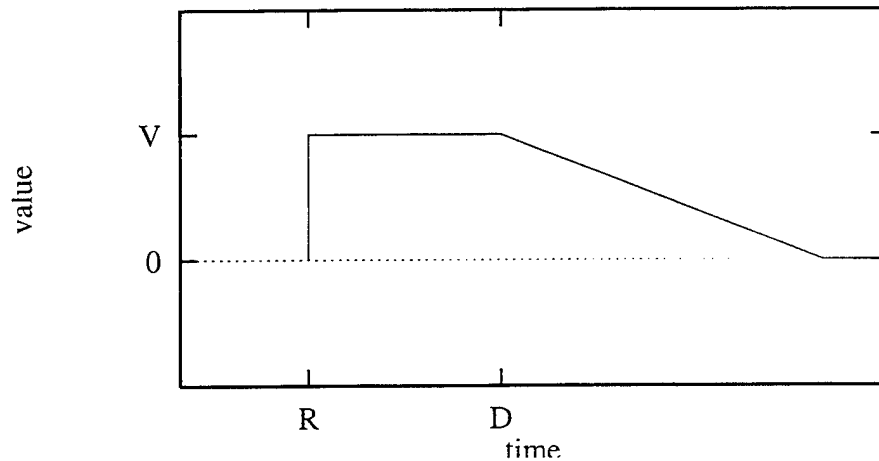


Figure 2-10: Soft Deadline Value Function

A non-real-time task might be described by the value function shown in Figure 2-11. In this case, completion of the task always has a positive value associated with it. This indicates no explicit timing constraint, although in practice, few of us are willing to wait indefinitely for a computation to complete.

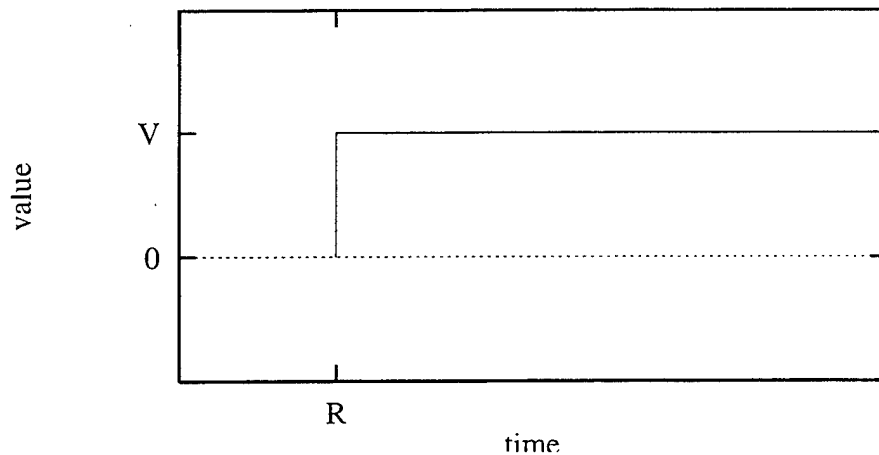


Figure 2-11: Non-real-time Value Function

2.2 Applications timing requirements

Many multimedia applications have real-time constraints. Even simple playback applications have real-time constraints that must be satisfied. For example, an audio player application might repeatedly read audio data from a file on disk and then enqueue the data for the audio device. Figure 2-12 illustrates the periodic computational requirements of such a playback application. The activity of the player is illustrated on the line labeled “P”, and the activity of the device is represented on the line labeled “D”. In each period, the applications reads, processes, and enqueues the data to a device. At the end of each period, the device reads the data out of its buffer, performs D/A conversion, and the analog signals goes to a speaker.

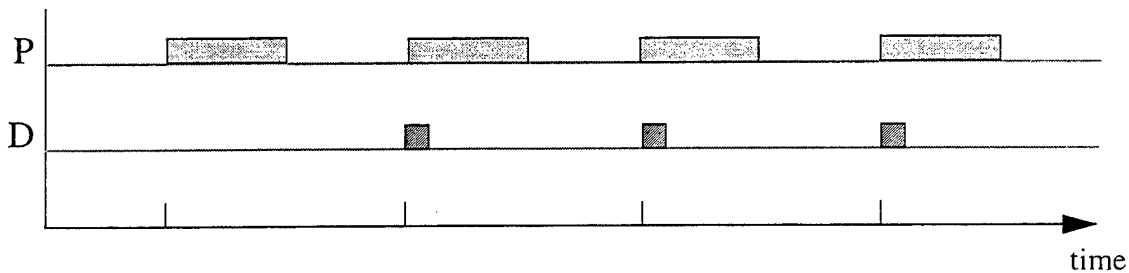


Figure 2-12: Periodic Playback Computations

A potential problem is that the computation of the audio player may be delayed so much that the device buffer empties and there are no samples for the device to convert to an analog signal. For a playback application, one solution is to introduce a large buffer and allow the playback application to execute for many periods to build up a large buffer of data ready to be played by the device.

Figure 2-13 shows a player with execution history shown on the line labeled “P”. The player buffers a number of data blocks for a device; the size of the data is indicated in the area labeled “B” between P and D. The device consumes data from the buffer, and the activity of the device is indicated on the line labeled “D”. Even if the audio player is delayed for a period or two, there will still be plenty of data in the buffer and the device will not run out of data blocks. The player can catch up with the processing that was delayed.

Another potential problem is that if some other activity on the machine is very active and manages to deprive the audio player of the resources (like processor time) for a very long time, then audio playback will be noticeably disturbed. This kind of intense competing activity can be avoided, and with large buffers audio playback will be quite smooth.

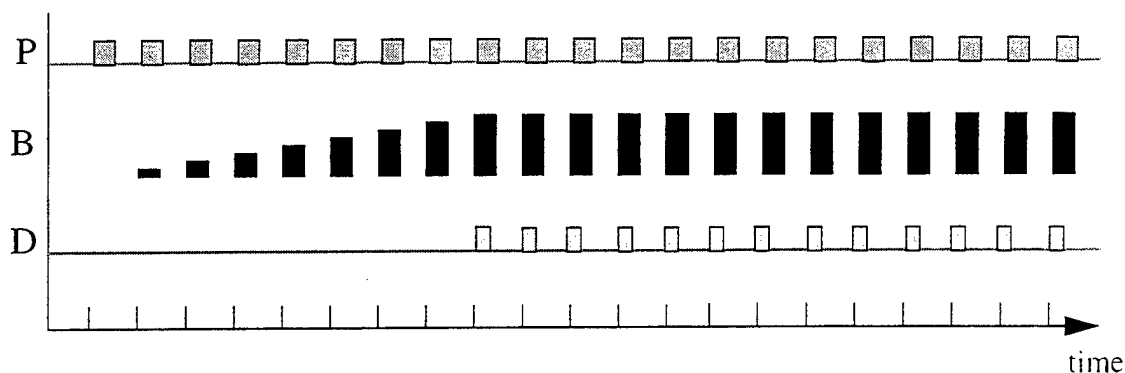


Figure 2-13: Playback Application Computing Ahead

Interactive applications cannot afford to use large buffers to smooth variations in scheduling delay. The delay introduced by large buffers is often too great to satisfy stringent end-to-end delay bounds in interactive applications like video teleconferencing. Instead, the systems must support applications that can ensure that bounds on the scheduling delay are observed. Thus the variations can be reduced and less buffering is required.

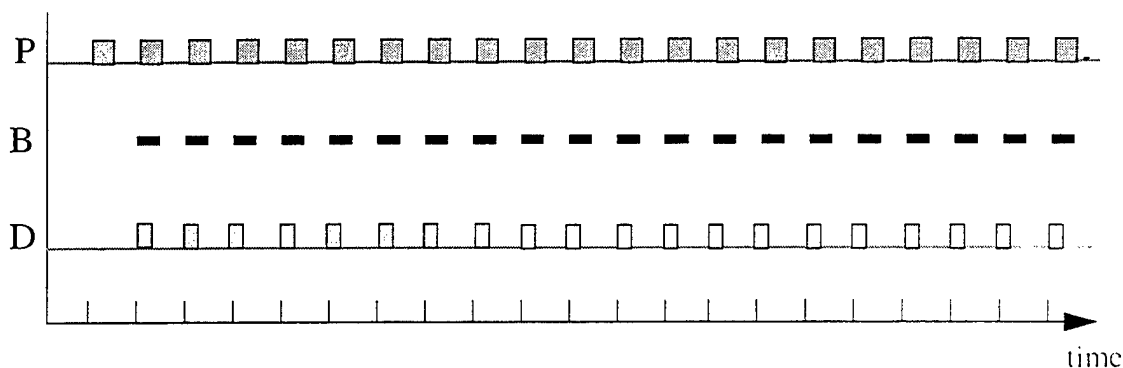


Figure 2-14: Interactive Application with Limited Workahead

Figure 2-14 shows the case where an interactive application can buffer the data generated in one period, but since there is a delay constraint, the application cannot afford to compute ahead several periods as in the previous case. The buffer must remain small. In order to make sure that the buffer is never empty when the device goes to get the next block of data, the application must make sure that the computations to enqueue the next block of data are done in time. This means it must also make sure that resources it will need from the system are available in time as well.

Other multimedia applications have timing constraints that differ from the periodic timing of the stream-oriented applications described thus far. For example, computer music applications involve performing computations to generate musical notes and other musical events. The number of notes that need to be generated at any point in time may vary widely depending on what the music calls for, so there is little periodic structure to describe the computations to be executed.

Silence-suppressed audio presents a similar problem. For audio conversations, it is not always necessary to transmit data for the silent portions of the audio streams in a two-way conversation. Again, the computational requirements become aperiodic when the amount and timing of data depend on speech patterns.

There are other applications, such as compressed video players, where a periodic execution pattern exists but where the computation time required within the periods varies. Compressed video data contains frames whose size varies according to the compression algorithm and the characteristics of the scene and how fast scenes are changing in the video source.

2.3 Quality of service management

Many multimedia applications have timing requirements and other quality of service (QOS) parameters that represent the user's desires and expectations for the performance of the applications. The complexity of providing for these timing requirements at the system level is exacerbated by the fact that the user may change those timing requirements at any time during the execution of the applications; and of course the user may create and terminate multimedia applications at any time.

2.3.1 QOS background

In recent years, researchers in the computer networking and in the telecommunications communities have been working on ways to express the QOS requirements for multimedia applications. Some of this work dealt with human perception requirements for various media [30,117], and other work focused especially on parameters and QOS architectures that are important in the context of scheduling traffic on a network [14,88]. This work can be considered an extension of the earlier work done in the telephone companies to characterize quality of service for telephone customers [100].

As researchers gained more experience with the idea of building networks that could provide quality of service levels suitable for different types of multimedia traffic, the question of how to ensure quality of service levels for end-to-end applications arose. Achieving that goal means QOS requirements must be supported in the operating system as well as the network. This observation was an initial motivation for the work described in this dissertation [76], and other system designers have started to focus on this aspect as well [3,21,46,53,83,102,126].

2.3.2 Mapping QOS parameters

In dealing with QOS management it is important to realize that there are different types of QOS parameters for different levels of the system. Applications must interact with the user in terms of user-level QOS parameters. For video, these user-level parameters might include frame resolution (width and height of each frame), number of bits/pixel, frames per second, maximum delay, and maximum jitter. For audio the user-level parameters might include sample rate, sample size, maximum delay, and maximum jitter. These are the types of parameters that might be meaningful to the user of a multimedia application. Or it might be preferable to offer QOS levels with names like: “worst”, “fair”, “good”, “better”, and “best” to simplify the interface. It would then be up to the application to translate these abstract QOS specifications to frame rates and sample rates.

Once the user-level QOS parameters are determined, they have to be mapped into system-level QOS parameters that would be meaningful for system-level resource management mechanisms. These system-level QOS parameters would describe how much time is needed on various resources. They depend on the user-level QOS parameters and on detailed computations that the application performs on data elements in the media stream.

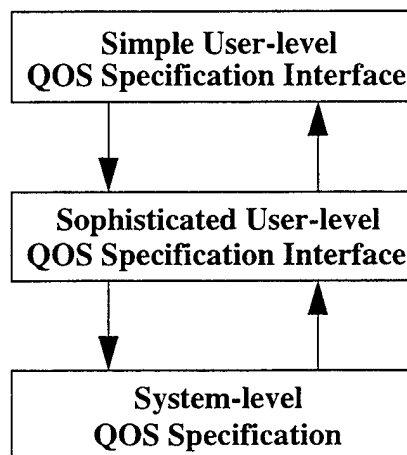


Figure 2-15: Levels of QOS Specification

Figure 2-15 summarizes these levels of QOS specification. The arrows in the figure indicate that there are mappings from one level to the next lower level and also that there are inverse mappings that come into play as well.

To allow the user to specify the QOS parameters desired at the highest level, the application must be able to map from user-level QOS parameters to system-level QOS parameters. The system-level parameters are required for the application to be able to ask for the resources it will need to execute. If the resources are unavailable, the system-level resource management mechanism should be able to communicate the fact that those parameters cannot be guaranteed. It should then initiate a negotiation to arrive at a set of system-level

parameters that can be supported by the system. The inverse mapping to user-level QOS parameters should yield a QOS specification that can be tolerated by the user. Thus, the inverse mapping from system-level to user-level QOS parameters is just as important as the forward mapping.

2.3.3 QOS negotiation

The user's QOS requirements may sometimes conflict with resource usage limitations, and therefore the QOS layer may need to negotiate user requirements to resolve such conflicts. This negotiation process may be needed throughout the lifetime of certain applications since user-level QOS requirements may change over the course of execution.

The approach advocated in this dissertation for handling the complexity of a dynamic execution environment (where programs may have changing real-time requirements) is to divide the problem into two parts. One part is the dynamic negotiation of resource allocation. The second part is the resource reservation and scheduling based on the allocation. These two parts can be addressed with a layering of functionality in the system where a system-level QOS management layer handles the resource allocation policy decisions and a low-level operating system resource reservation mechanism handles the details of dynamic scheduling and usage enforcement.

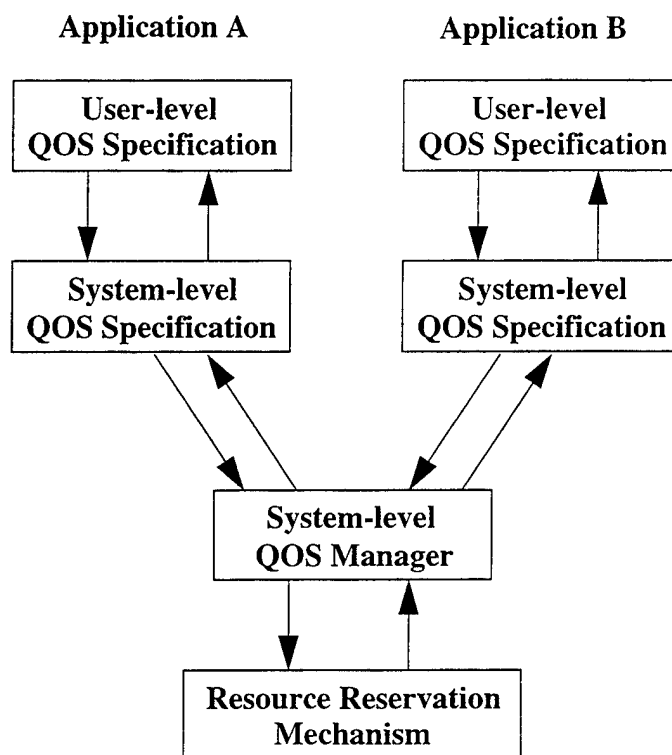


Figure 2-16: QOS Levels with QOS Manager

An illustration of the basic outline of the QOS management system appears in Figure 2-16. Each application has user-level QOS specifications. The applications map the user-level QOS parameters in to system-level QOS parameters and then negotiate with the system-level QOS manager to determine a mutually acceptable set of system-level parameters. The operating system contains a resource reservation mechanism which is used by the system-level QOS manager to actually allocate the resource capacity, schedule appropriately, and enforce the resource reservations.

The QOS management layer makes policy decisions about where resource capacity in the system should be allocated. To do this, it will depend on input from applications as they make their system-level QOS parameters known. The QOS manager may take input from user preferences expressed in the form of rules about which applications are more important than other applications, and it may take input from user interface tools designed to help the user manage resource allocation in the system. The QOS manager may also coordinate with other QOS managers on remote hosts for setting up distributed multimedia applications that require resources on several different hosts.

2.4 System design approaches

Several approaches have been used in the past to support interactive and playback types of multimedia applications. These range from hoping for the best, to dedicating expensive resources, to system support for real-time programming.

2.4.1 Specialized hardware

One approach to supporting real-time multimedia applications is to dedicate hardware to the tasks that must be performed in real-time. This relieves any contention for resources. As an example, Pandora's Box [44] was an early attempt to support multimedia applications in the context of a desktop workstation. The box contained six transputers, each one dedicated to a particular activity or class of activity including audio processing, storage/disk management, video processing, decompression, network communication, and bus management. This box was connected to a Sun workstation, to a network, and to a monitor. It coordinated graphical display from the workstation and video streams from the network or other devices, combining them and displaying the result on the monitor. The system allowed researchers to learn much about programming multimedia applications, what kinds of applications were useful, and user interface issues. However, the cost of the box was very high, and the complexity of programming the box itself was also great.

2.4.2 Time-sharing systems

A number of multimedia applications are available for personal computers and desktop workstations that run more sophisticated operating systems like UNIX and Windows NT. These systems use time-sharing scheduling policies that are not particularly well suited for meeting the timing constraints of multimedia applications.

Time-sharing schedulers are tuned to provide fair allocation of resources among users which are considered equally important. These schedulers also look at whether a processes is compute-bound or not, and they depress the priority of compute-bound processes in favor of interactive (or I/O-bound) applications which can benefit from better response times.

This kind of scheduling behavior works well in mainframe systems, but may work against multimedia applications. For example, a video application that performs a filtering computation on video frames may look compute-bound to a time-sharing scheduler and may therefore get a low priority compared to network and I/O activity on the machine. This may occur even if the video application is the most important activity to the user.

Consider a teleconferencing application that display several video streams on the screen at the same time. A fair time-sharing scheduling algorithm would give each of these streams an equal share of the processor, resulting in the same frame rate for all of the video displays. This might not be appropriate for the particular application. The user might want more control over where resources are focused, perhaps to show a higher frame rate for the person in the conference who has the floor.

In many operating systems such as UNIX [62], VMS [80] and Windows NT [24], the time-sharing scheduling policy is augmented with a fixed priority extension. The extension is usually in the form of a range of fixed “real-time” priorities. With fixed priorities, it is possible to exercise more control over how the processor is scheduled, but there are other problems. Many of these issues arise in the context of real-time operating systems as well, where fixed priority scheduling is commonly used. The next section addresses some of the difficulties of real-time programming with fixed priority schedulers.

2.4.3 Real-time operating systems

Much work in recent years focuses on how to apply real-time systems techniques to multimedia systems and applications. This includes work directed at methods for using technology available in commercial real-time operating systems as well as efforts to build research prototype operating systems.

2.4.3.1 Commercial real-time systems

Most commercial real-time operating systems support fixed priority (FP) scheduling of processes [99,104]. FP schedulers while useful for real-time scheduling, cannot by themselves support multimedia applications.

For example, fixed priority schedulers have no mechanism for dealing with overflow situations. In general, real-time operating systems do not have the mechanisms for deciding whether enough resources are available in a system to run a new application; they do not have support in the system for admission control. Furthermore, there are no mechanisms for detecting and dealing with overload situations when too many applications are allowed to run. These issues of load management are typically addressed in an ad hoc manner by sub-systems specific to particular applications.

Even if a system user could determine that a particular collection of applications could run successfully on a system, the problem of determining what priority assignment should be used remains unsolved. An application designer may use multiple threads or processes in the implementation of the program, and that designer will undoubtedly know enough about the processes' computational requirements, timing constraints, and precedence relationships to assign priorities in a reasonable way. However, when running several such applications developed by different people on the same system, the question arises: How should priorities be assigned to processes in different applications in a way that will result in correct timing behavior? Without global knowledge of all processes and their timing constraints, assigning priorities appropriately is exceedingly difficult.

In practice, commercial operating systems are used mainly in embedded applications where designers carefully measure the resource requirements and coordinate scheduling based on a scheduling analysis of the specific task set designed for the application [68]. The system designers have global knowledge about resource requirements, and they use that information in the scheduling analysis to generate a priority assignment. This makes the systems rigid and difficult to maintain. Much more flexibility would be desirable.

2.4.3.2 Research prototype real-time systems

Several research prototype operating systems have applied results from real-time scheduling theory to multimedia applications [3,53,126]. The DASH kernel [3] used an admission control algorithm based on a timeline and then used earliest deadline scheduling to actually sequence the tasks. Other systems used analyses from real-time scheduling theory to guarantee timing constraints for applications. For example, YARTOS [53] uses algorithms for scheduling sporadic tasks [52] to guarantee timing constraints. In order to guarantee performance, the computational requirements of the applications must be measured and analyzed along with the timing constraints such as delay bounds. Then the application can be run with the expectation that timing constraints will be satisfied.

In RT-Mach [125], much of the work on support for multimedia applications (other than the work described in this dissertation) used the traditional rate monotonic scheduling algorithm. As with YARTOS, the computational requirements of applications were measured in advance and analyzed to ensure that timing constraints would be met. The applications could then run successfully on the system. Much of the work on RT-Mach centered on high-performance real-time threads packages [92] and QOS managers [127].

These systems took a careful approach to analyzing and guaranteeing timing requirements for multimedia applications based on real-time scheduling analyses. They also incorporated advanced real-time system features such as priority inheritance protocols [108] and inheritance protocols for deadline scheduling [16]. These features are essential for supporting strong real-time guarantees among programs that share data and interact in other ways.

2.4.4 Soft real-time system support

Several multimedia systems have used scheduling algorithms like earliest deadline first to make the system more sensitive to timing without necessarily guaranteeing that timing

constraints will be met. For example, a system based on Chorus [103] proposed using deadline scheduling with no admission control [21]. Other algorithms such as lottery scheduling [133] attempt to support multimedia applications using proportional sharing of resources without real-time delay guarantees. A deterministic version of the approach called stride scheduling [134] was proposed to better support multimedia applications.

These types of systems are able to be more sensitive than time-sharing systems to the timing constraints of multimedia applications, but without effective admission control, overload cannot be prevented.

2.5 Reserve abstraction

The reserve abstraction described in this dissertation addresses several of the key problems raised above. The abstraction provides a framework for reasoning about resource reservation in an operating system. Other research efforts have focused on reservation of different resources in isolation. A framework to unify various reservation algorithms is needed.

The reserve abstraction gives resource reservation first-class status in the operating system. Reserves can be allocated independent of any particular process, and references to reserves can be passed around and bound to different processes as appropriate. For example, a real-time client might pass information about its timing constraints to a server to ensure expedited service. This is in contrast to the approach where timing constraints and resource usage requirements are associated directly with processes which makes it difficult or impossible to have one process temporarily take on the timing constraints of another process.

The key feature of the reserve abstraction is the enforcement mechanism. This prevents applications from overrunning their reservations if that would interfere with the timing requirements of other reserved activities.

The reservation framework and first-class status of reserves provide the power and flexibility to deal with the problems that arise in real-time system design and practical resource management. And the enforcement mechanism guaranteed proper resource scheduling. In combination, these aspects of the reserve abstraction offer an effective solution to the general real-time programming problem.

Chapter 3

Reserve Model

This chapter gives a definition of reserves on resources, which form the basis of the reservation model. Reserves provide a framework for integrating admission control, scheduling, and usage enforcement. Issues in reserve management are also addressed.

3.1 Reserve abstraction

The reservation model defines the concept of a *reserve* against a particular operating system *resource*. A reserve is a first class object that represents a part of the capacity or a quantity of the resource that is set aside for a computation which presents that reserve along with a request to use the resource. The word “capacity” is used in a broad sense here: reserving a portion of the capacity of a resource means that a thread will have access to the resource subject to some detailed reservation parameters, and the parameters are specific to the resource and the reservation system implementation. As an example, a reserve might specify that 30 ms of computation time on the processor are reserved out of every 100 ms of real time.

Reserving resource capacity implies that the resource can be multiplexed among several computations, and the model focuses on multiplexed resources such as processors and network bandwidth. Other types of resources such as physical memory pages and buffers are not amenable to extremely fine-grained multiplexing, and these are referred to as discrete resources. In this model, discrete resources are reserved on a per unit basis and the reservation dedicates the resources units indefinitely rather than implying a multiplexed usage of capacity over time.

3.1.1 Reservation guarantee

A key requirement in offering a resource reserve abstraction in a system is that the reserved resource capacity be available, subject to the reservation parameters, to a computation which presents the reserve and requests the resource. If the system cannot guarantee

that the resource capacity will be available as promised, the usefulness of the system is limited. Therefore, the enforcement of resource reservations is of critical importance. Enforcement is important not only to protect against malicious users, which may present a problem in systems where resources are shared by many, but also to relieve individual applications from the burden of ensuring that their own performance is strictly predictable and controlled. The system should tolerate applications which may try to use more than their reservation allows, isolating unrelated applications from this kind of behavior. This kind of *temporal isolation* is similar in concept to the isolation provided by a virtual memory system, which allows a process to try to access memory locations as it wishes, intervening only when a memory access is not allowed. In no case should a virtual memory system allow a process to access the memory of another process's protected memory, and likewise in no case should a reservation system allow a thread to impinge on the reserved resource capacity of another thread.

Thus, the system guarantees that resource capacity, given by the reservation parameters, will be available to the thread that has a reserve. However, it is up to the application programmer to make sure that the thread is in a position to take advantage of the resource capacity when it is made available. The reservation system itself makes no claims that a particular application will meet its timing constraints. For example, if an application blocks indefinitely waiting for a message, it may not be in a position to take advantage of resource capacity when it is available. For an application to have predictable real-time performance, it must have the proper resource reserves, and it must be able to use those resource reserves in a way that satisfies the timing constraints of the application.

3.1.2 Scheduling frameworks

The reserve abstraction can accommodate different frameworks for admission control, scheduling, and enforcement. Most of the features of reserves, specifically the operations available in the reserve abstraction, remain the same even if the scheduling framework changes. The primary differences in the interface to a different framework are the specification of the reservation request parameters for admission control and the resource usage statistics available from the enforcement mechanism.

3.1.3 Styles of programming with reserves

Reserves can be used in two different styles of real-time programming. Reserves support strict hard real-time applications and can equally well support more flexible soft real-time applications. The primary distinction between hard and soft real-time programming in the discussion of the reservation model is:

- whether resource usage requirements are carefully measured and specified in exact detail guaranteeing performance before the program is actually deployed (hard real-time), or
- whether resource usage requirements and resource capacity reservations are dynamically adjusted based on run-time usage measurements instead of being matched exactly during the design phase.

In either case, the resource reserves guarantee the requests that are admitted to the system, and whether or not those reserves are used for hard real-time programming or soft real-time programming depends on how the applications themselves use the reserves.

Another distinction in real-time programming using reserves is whether resources are reserved locally for each thread or whether they are reserved globally for an *activity* that may span multiple threads in different protection domains and even on different machines. In the activity-based model of using reserves, the originator of an activity acquires resource reserves for the activity and then passes those reserves along with any requests made to various servers. Using this model, accounting for resource usage across clients and servers is simplified, and the negotiation of quality of service parameters can be simplified as well. The reserves for each request come in with the request, and the server charges resource usage to those reserves when servicing the corresponding request. The responsibility of getting the appropriate resource reserves falls to the original client.

To summarize, several features of reserves are useful for both hard and soft real-time programming:

1. Take care of global admission control decisions, relieving the designer of doing global scheduling analysis.
2. Schedule threads on resources according to their reserves.
3. Accumulate usage information that could be useful during development, especially for adaptation in soft real-time applications.
4. Prevent interference from other real-time applications and non-real-time applications and activities that may be competing for the same resources.
5. And finally, reserves can serve either to separate the resource allocation and management of modules from each other or to integrate the resource allocation and management of modules, allowing reservations to span multiple threads and protection domains of a single activity.

3.2 Basic reserves

The *basic reserve* provides an abstraction for capacity on a particular resource. Note that this statement about basic reserves *does not guarantee that applications will meet their timing requirements*. The only guarantee is that *capacity will be reserved and available to be used*. We will explore the issue of what guarantees can be made at a higher level about the behavior of applications that use reserves.

The reserve itself is an operating system abstraction that is orthogonal to control structures like threads. A thread may bind to a resource reserve in which case the scheduler will use the information in the reserve when making scheduling decisions about the thread. Multiple threads may be bound to a single reserve, but typically a reserve will have only one thread bound to it at a time. The scheduler will always find an associated reserve, although sometimes that reserve will be a *default reserve* which has no actual reservation and just

serves to accumulate the resource usage of all threads that make unreserved use of the resource.

The specification of the reservation depends on the type of resource. Multiplexed resource reservations include information about the amount of work to be done per period of real time. They may also include a delay requirement. This parameter would specify the maximum amount of time after the beginning of each period the thread will have to wait before getting its reserved time on the resource. Discrete resources reservations just specify a count of the units of discrete resource required; they include no notion of time.

Despite the differences between multiplexed and discrete resource reservations, they share the same basic structure. They both require:

1. a reservation specification interface,
2. an admission control policy,
3. a scheduling policy, and
4. an enforcement mechanism.

For discrete reserves like memory pages and network buffers, the reservation specification gives a number of units of resource being requested. The admission control policy for discrete resources would just check the availability of the requested number of units of resource. Since discrete reserves are by definition not multiplexed, they require no scheduling.

It is important to note that the basic reserve abstraction is independent of the admission control and scheduling policies used. For multiplexed resources, reserves provide a framework to request resource capacity reservations, do admission control, schedule computations, and enforce capacity reservations. The choice of admission control and scheduling policies will impact the way reservation requests are specified and the way the enforcement mechanism tracks their behavior, but the framework is general enough to accommodate different policies. The following sections illustrate the reserve model in terms of a periodic scheduling framework.

3.2.1 Reservation specification

The reservation system must provide a way for applications to specify the resource capacity they would like to reserve. The form of the specification differs from resource to resource, and different admission control and scheduling policies may require different reservation specification parameters. In the most general sense, reservations specify a duration of usage with some time constraints be available, used, and replenished by some specific regimen.

As an example of the kind of parameters that might appear in a specification, a resource reservation may specify an amount of time to be spent on the resource per period of real time, and it may specify a start time for the periods as well. For example, a reservation request might specify 30 ms every 100 ms starting at 1:00pm.

Figure 3-1 illustrates how the reserved time might be consumed in a simple computation time per period of real time framework for reserves. The reserved computation time is available to be used during each reservation period. The computation time is guaranteed to be available at some point during the period; it is not guaranteed to be in any particular place such as the front of the period or the end of the period.

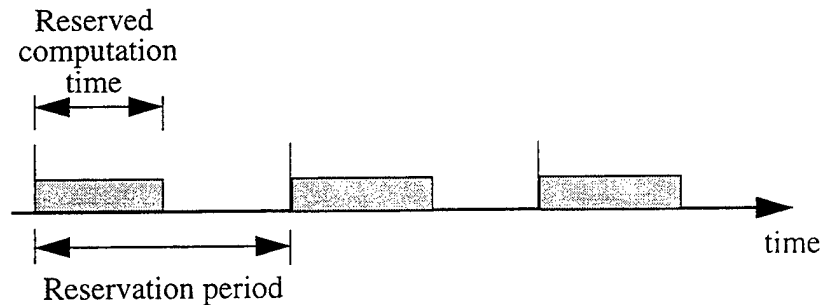


Figure 3-1: Periodic Scheduling Framework

There are many different scheduling policies and scheduling analysis techniques that could be used to provide a reserve abstraction, each of which would require a corresponding admission control test, scheduling policy, and enforcement mechanism.

3.2.2 Admission Control

An admission control policy associated with each resource decides which reservation requests for that resource can be admitted and which must be denied. It makes this decision based on the parameters provided in the reservation request and information about the other reservations that have already been granted for that resource. The admission control policy necessarily depends on the scheduling policy in order to do an admission control analysis.

3.2.3 Scheduling

The scheduling algorithm for a resource makes decisions about the order in which threads receive time on a resource. The scheduler looks at the reserve owned by each thread that is ready to run, and uses information in the reserve to determine which thread will get access to the resource. The algorithm supports the decision made by the admission control policy.

The scheduler must also coordinate with the enforcement mechanism to make sure that it does not try to schedule threads associated with a reserve that has already used its reserved resource time for a particular reservation period. Thus a reserve which still has time left on its reservation is in “reserved mode” and one that has run out of time is temporarily in “unreserved mode.” This represents a significant departure from other real-time scheduling algorithms, which generally assume that the resource usage requirements of application are accurately characterized and need not be enforced [67].

3.2.4 Enforcement

The reservation system must ensure that processes do not use more than their reserved capacity or reserved units of a resource. Enforcing reservations on discrete resources is straightforward; the system ensures that a resource dedicated to one process is not re-allocated to another process. Enforcing multiplexed reservations requires the system to keep accurate usage numbers that describe how much capacity has been consumed against each reservation. If a thread attempts to use some capacity beyond its reservation, the system must recognize this and actively prevent the process from consuming any additional capacity in reserved mode (consuming additional capacity in an unreserved mode may be allowed.)

If for some reason the reserved time on a resource is not used by the owner of a reserve in a given reservation period, that allocation of time is lost to the owner. The owner may not be in a position to use the resource if it is blocked waiting for some other resource to become available or for synchronization or communication with another computation. The resource will not necessarily be idle for that amount of time since the scheduling policy is free to allow an unreserved computation to use the resource (as long as the unreserved computation can be preempted to allow the reserve owner to use the resource). This implies that a computation may not save up reserved time (by not using it) and then use it all at once in a burst. The allocation of resource time is available during each period, but cannot be carried over past the end of the period.

On the other hand, if the thread that owns a reserve consumes the entire reserved allocation for a reservation period and attempts to continue executing, the thread will compete with the other unreserved computations for the resource under whatever policy the resource scheduler uses for unreserved computations. Thus a reserve may be in *reserved mode* where it still has some resource usage allocation left for the current reservation period, or it may temporarily be in *unreserved mode* where the allocation for the current reservation period is depleted (until the next reservation period).

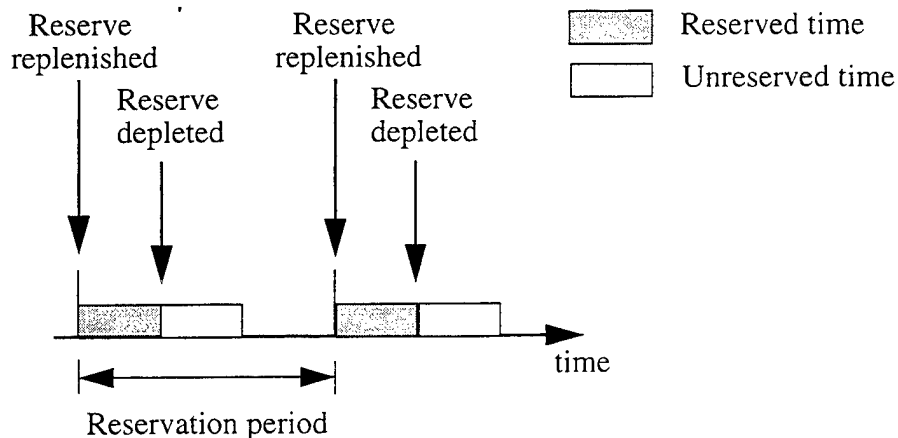


Figure 3-2: Enforcement Illustration

Figure 3-2 shows the reserved time made available on a particular resource for a single reservation. At the beginning of each reservation period, the allocation of reserved time is replenished, and the thread that has this reservation uses the resource in “reserved mode.” After the reserved time allocated for that period is depleted, the enforcement mechanism generates a scheduler event to indicate that the reserved time has been consumed for that period. The scheduler is responsible for using that information in making scheduling decisions. In the figure, the thread continues to use the resource in “unreserved mode,” consuming more time on the resource at the discretion of the scheduler. The execution history shown in the figure is based on the assumption that no other threads compete for the resource and that the policy lets it run in timing-sharing mode after its reservation has expired, and so the thread can get time in unreserved mode. At the beginning of the next reservation period, the reserve is replenished and the thread can run in reserved mode again. The main point of this figure is that the enforcement mechanism tracks resource usage and raises this “reserve depleted” event for the scheduler. The scheduler can then use this information in making future scheduling decisions. For example, it can give other reserved activities preference or allow unreserved time-sharing activities to use the resource.

Three important issues arise in the design of the enforcement mechanism:

1. how to accurately accumulate resource usage,
2. how to notice that a thread has depleted its reserve for a resource,
3. how to know when to replenish the allocation of a reserve.

To accurately accumulate the resource usage for each reserve, the system records the time during each reserve switch. A reserve switch occurs in two cases: when a thread context switch is performed or when a thread is changing the reserve against which it will charge its computation time. In the case of a thread context switch, the reserve switch records the current time, cancels the overrun timer (which signals the reserve depletion event as described below), computes the time the old thread ran, and adds that time to the accumulated usage of the old thread. The reserve switch mechanism then saves the current time for use later in computing the computation time of the new thread and sets the overrun timer. A reserve switch triggered by a thread changing the reserve to which it wants to charge its computation time works the same way, the only difference is how the reserve switch is triggered. Figure 3-3 illustrates how timers are used in enforcement. In the execution history on the timeline, it shows the reserved activity of interest in a solid pattern and some other activity (associated with other reserves) in a striped. The reserve switches (also context switches in this example) between these activities occur where the striped boxes and the stippled boxes meet.

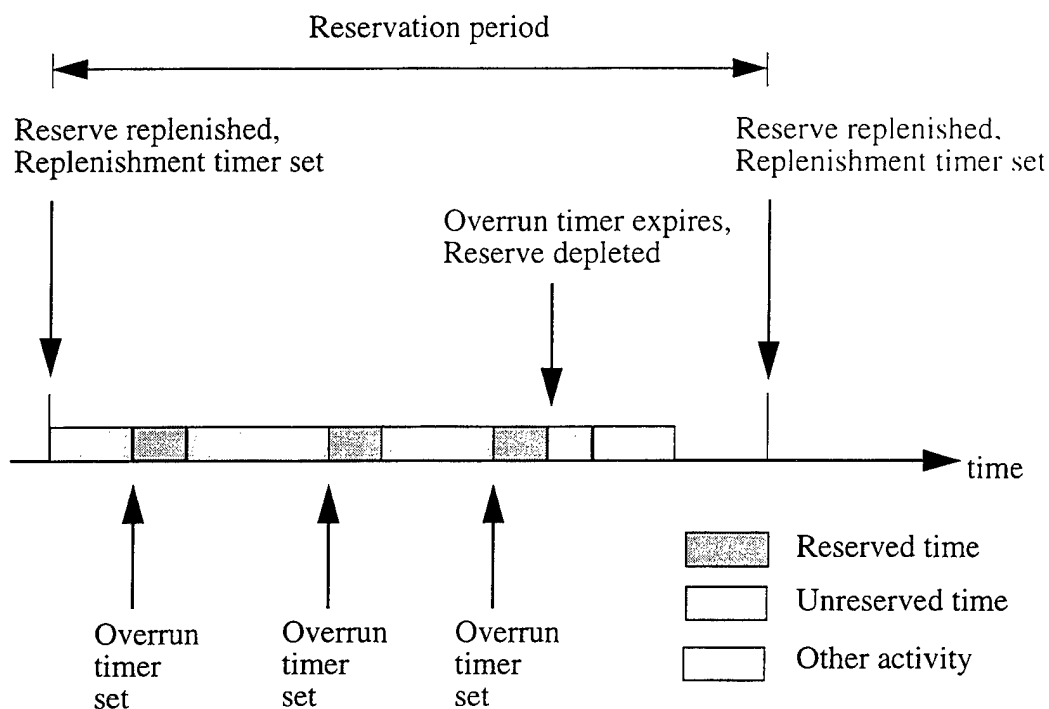


Figure 3-3: Enforcement Timers

The overrun timer is set during the reserve switch to expire at the end of the new thread's remaining reserved computation time or at the end of its reservation period, whichever is earlier. If the new thread is preempted before its reserved computation time is completed, the overrun timer will be cancelled. If the new thread consumes all of its reserved time, the overrun timer expires, and the scheduler is called to take some action based on that event. Figure 3-3 shows where the overrun timer is set for the reserve of interest; the overrun timer may also be set for the other activity if it is reserved, but that is not shown in the figure. The overrun timer in the figure does not actually expire until the last time it is set.

The handle replenishment, each reserve has a replenishment timer which is started at the reserve start time and which expires periodically at each reservation period. The replenishment timer records the usage accumulated on the corresponding reserve at the time of the reservation period and records that along with the current time as a "usage checkpoint." Then the timer handler changes the state of the reserve to reflect a new allocation of resource usage and resets the periodic replenishment timer. This replenishment model corresponds to a deferrable server [120] replenishment scheme; other replenishment methods are described and analyzed by Sprunt [111,112]. Figure 3-3 illustrates where the replenishment timer is set relative to the reservation period.

3.3 Reserve propagation

One important feature of the reserve model is that reserves can be passed from clients to servers, enabling the server to take advantage of the resources the client reserves for its entire computation. Passing reserves also enables the server to charge the resource usage it consumes to the appropriate client, preserving system-wide consistent usage accounting. This is called *reserve propagation*.

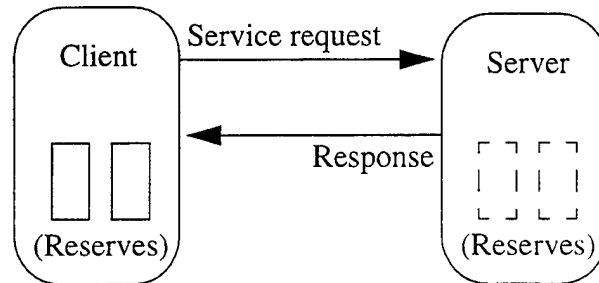


Figure 3-4: Reserve Propagation

Figure 3-4 illustrates a client/server interaction with reserve propagation. Assume that the client acquires resource reservations sufficient for the computation that it will perform locally as well as the computations to be done by servers on the client's behalf. In the figure, these reserves appear in the clients as two small rectangles. The interaction is a straightforward remote procedure call from the client to the server. For simplicity, assume that the client sends an RPC request to the server and waits for the reply. The server processes the request and sends an RPC reply, and then the client receives the reply.

When the client send the service request, it also sends references to the reserves that it has allocated. These reserves are to be used by the server as it processes the client's request. Thus, the server must start charging its resource usage to the client's reserves when it starts processing that request, and it must stop charging to those reserves when it finishes with the request and sends back the response.

Ideally, a server would schedule service requests to execute in the same order that the computations would execute if the clients could do them instead of the server. For example, the processor scheduler orders ready threads based on the processor reserve parameters. This ordering can be seen as a sort of "priority" ordering among those activities. If a thread makes a request of a server, the server should take the "priority" in the scheduler's ordering while it is servicing that request. Then the fact that a client relies on a server for some computation does not affect the progress of its computation with respect to other threads.

To help the server achieve this ideal, the RPC mechanism should propagate the reserve "priority", as represented by the reserve and its reservation parameters, of the client to the server. The queue of service requests for the server must be maintained in reserve "priority"

order, and a kind of “priority inheritance” must be used to prevent priority inversion in the access to the server.

On the receipt of a new service request, his “reserve priority inheritance” mechanism enqueues the request in the priority queue, and then it checks to see if the thread that will service the requests is waiting for new requests or servicing a previous request. If the thread is busy and the “priority” of the new request is greater than the “priority” of the currently processing request, the RPC mechanism sets the priority of the thread so the priority of the new request. It does not, however, change the reserve that the thread is charging against. When the server thread finishes the previous request and receives the new request, it keeps the priority of that new request (which it inherited before) and also begins charging to the reserves associated with the new request. When the request is finished, the server thread stops charging against the client’s reserves.

The “priority” inheritance mechanism described here, which is referred to as “reserve propagation”, differs from traditional priority inheritance in two ways:

1. reserve propagation specifies how a server should change the reserves it charges to based on the client it is servicing whereas traditional priority inheritance has no notion of charging to a reserve or account,
2. the “priority” of the server may change during the course of the request processing if the reserved resources are depleted during that time whereas with traditional priority inheritance, the priority is fixed.

The fact that a server’s “priority” may drop during request processing complicates reserve management and reserve “priority” inheritance. When a reserve is depleted, the scheduler must determine whether the thread charging against the reservation inherited the reserve “priority” or not. If not, the thread’s order in the ready queue may change. If so, the scheduler must determine from the threads pending request queue what the appropriate reserve “priority” for the thread should be, given the change in the state of the reserve that was depleted.

Reserve propagation from client to server is not mandatory. The next chapter discusses different programming models where this is useful and where it is not. Briefly, reserve propagation is useful when the system is organized such that an application allocates the resource reserves it will need for all of its activities and passes those reserves around to the servers it invokes to do work on its behalf. In this model, applications need not negotiate quality of service parameters explicitly with these servers. The scenario where reserve propagation is not that useful is where the system is organized such that applications negotiate quality of service explicitly with all of its servers.

3.4 Example scheduling frameworks

Many different admission control and scheduling policies could be used to support the reserve abstraction. For example, reserves based on rate monotonic scheduling [67] would be able to guarantee the availability of a certain amount of time on a resource per period of real time. For pure rate monotonic scheduling, the delay associated with receiving the com-

putation time in each period would be the length of the period itself. For deadline monotonic scheduling [64], the delay bound could be shortened. The following sections discuss these frameworks and others in more detail as they apply to the reserve abstraction.

3.4.1 Rate monotonic

The rate monotonic (RM) priority assignment of Liu and Layland [67] can guarantee that a certain amount of computation time will be available for a reserve for each period of real time with a delay bound equal to the length of the period. The discussion of rate monotonic scheduling uses the word "task" to denote the series of instances of a computation; with reserves, it is understood that this computation represents available resource capacity and not necessarily a complete program. Under rate monotonic scheduling, higher priority is assigned to the higher frequency programs. The rate monotonic scheduling analysis yields a basis for a processor reservation admission policy.

3.4.1.1 Reservation parameters

Reservation parameters for the simplest form of rate monotonic based reserves include computation time and reservation period: A start time is also useful for controlling the phase of the periodic reservations. This allows the programmer to synchronize the availability of the reserved computation time with a periodic program.

3.4.1.2 Admission control decision

Let n be the number of periodic tasks and denote the computation time and period of program i by C_i and T_i , respectively. Liu and Layland proved that all of the tasks would successfully meet their deadlines and compute at their associated rates if

$$\sum_{i=1}^n \frac{C_i}{T_i} \leq n(2^{1/n} - 1)$$

When n is large, $n(2^{1/n} - 1) = \ln 2 \cong 0.69$. This bound is pessimistic: it is possible for task sets which do not satisfy the inequality to successfully meet their deadlines, but this cannot be determined from the Liu and Layland analysis.

An admission control policy follows naturally from this analysis. To keep track of the current reservations, the system must remember the utilization of the tasks that have reserved computation time, and the total reservation is the sum of these individual utilizations. A simple admission control policy is to admit a new reservation request if the sum of its utilization and the total previous reservations is less than 0.69. Such a policy would leave a lot of computation time that could not be reserved. One possibility is to use that time for unreserved background computations. Another possibility is to use the exact analysis of Lehoczky *et al.* [63] to determine whether a specific collection of reservations can be scheduled successfully, although the exact analysis is more expensive than the simple, pessimistic analysis above. In their work, Lehoczky *et al.* also gave an analysis showing that on aver-

age, task sets can be scheduled up to 88% utilization. So in most cases, this unreservable computation time is only 10-12% rather than 31%.

It should be noted that the rate monotonic scheduling algorithm was analyzed under simplifying conditions. Liu and Layland [67] made the following assumptions to enable their analysis:

- arrivals are periodic, and the computation during one period must finish by the end of the period (its deadline) to allow the next computation to start,
- the computation time of each program during each period is constant,
- computations are preemptive with zero context switch time, and
- computations are independent; i.e. computations do not synchronize or communicate with other computations.

In the context of the reserve abstraction, this means that rate monotonic scheduling can be used to guarantee that resource capacity is available to the applications. However, applications that have precedence constraints with other applications may not be in a position to use the available resources.

3.4.1.3 Scheduling

Reserved mode activities get precedence over unreserved. Among reserved mode activities, smaller period gets precedence over larger. Among unreserved activities, some time-sharing algorithm may be in effect.

3.4.1.4 Enforcement

Accumulate usage in each period. Update usage (determined using accurate measurement techniques) at each context switch. Set a timer for the currently running activity to expire at the end of its reserved usage. Set another timer for each reservation period.

3.4.2 Deadline monotonic

The deadline monotonic scheduling (DM) algorithm [7,64,66] is closely related to the original rate monotonic (RM) algorithm [67]. DM has the same kind of periodic scheduling frame as RM; the difference is that with DM, there is an additional parameter called the deadline specifies the duration of time by which the computation released at the beginning of the period must be completed. For the original version of RM, this deadline is assumed to be the end of the period, when the next instantiation of the computation will be released. For DM, this deadline is specified explicitly.

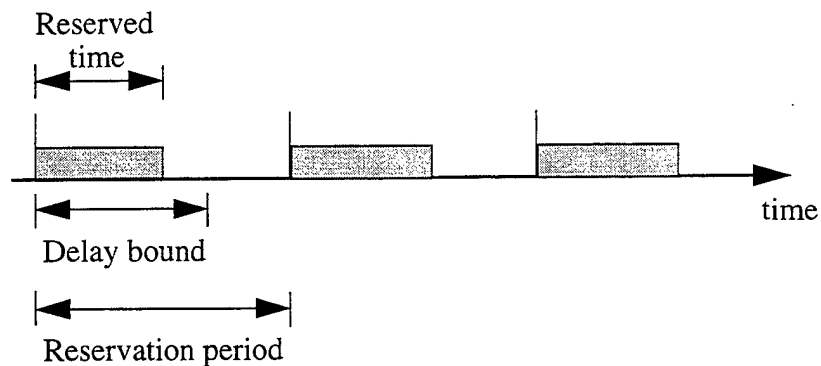


Figure 3-5: Deadline Monotonic Scheduling Framework

Figure 3-5 illustrates the periodic scheduling framework of DM along with the additional deadline parameters that does not appear in the original RM algorithm. The deadline in this case is before the end of the period, but it could also be after the beginning of the next period (in which case there would be multiple instantiations of the computation, started in different periods, at one time).

3.4.2.1 Reservation parameters

The reservation parameters for DM are the same as RM with the addition of the deadline parameter. As an example, a reservation request may specify 30 ms on the resource be reserved for every 100 ms with the delay constraint that the 30 ms must be available within 50 ms of the beginning of each 100 ms period. As for rate monotonic scheduling, a start time parameter is useful for synchronizing reservations with periodic threads and with other reservations.

3.4.2.2 Admission control decision and scheduling

Schedulability analysis tests for DM are given by Lehoczky [64] and by Audsley et al. [7]. These tests are quite a bit more complicated than the simple schedulability bound test for RM, involving systems of equations that have to be checked. Even so, the schedulability tests provide suitable admission control decisions for a reservation mechanism based on DM.

Scheduling is based on the deadline monotonic algorithm that assigns fixed priorities to tasks based on the deadline value. Shorter deadlines are assigned higher fixed priorities than longer deadlines. As with the rate monotonic algorithm, the reservation mechanism distinguishes between reserved mode activities and unreserved mode activities. At the beginning

of any given reservation period, an activity with a reservation is in reserved mode until it consumes all of the reserved time for that period. It is then changed to unreserved mode. The scheduler services reserved mode activities first, in order of their deadline values. If there are no reserved mode activities, unreserved activities are scheduled.

3.4.2.3 Enforcement

The enforcement mechanism accumulates usage in reserved mode until one of the following occurs: the resource usage reserved for that period is consumed or the deadline time has passed. In either case, the activity is changed from reserved mode to unreserved mode where it can compete with time-sharing activities for the resource.

3.5 Basic reserve types

Operating systems manage many different kinds of resources that system and user programs may use to do their work. The most important examples are processors, physical memory, buffers, and network bandwidth.

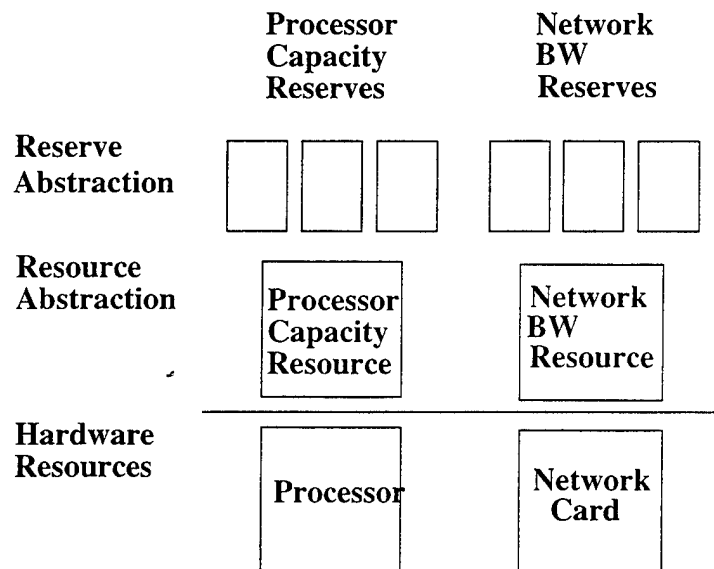


Figure 3-6: Resources and Basic Reserves

Many of these resources must be managed in the reservation system, so we define *basic reserves*, which are used to reserve and control the usage of different types of system resources. Each basic reserve type is associated with a resource type in the system. Figure 3-6 illustrates the relationship between the operating system resources and the basic reserves.

3.5.1 Processor

Processor capacity reserves represent reserved time on a processor. Reserve requests specify capacity in terms of time that will be reserved on the processor, rather than in terms of instructions that will be executed or any similar measure of processor usage. The requests may specify other information, depending on the admission control and scheduling policy in effect. The discussion in the following sections assumes a deadline monotonic scheduling framework where the reservation request specifies the amount of time to reserve on the processor, a period at which the allocation will be replenished, and a delay bound. These sections will cover these topics in more detail and will additionally discuss issues in enforcement, blocking time, and usage monitoring.

3.5.1.1 Units of work

Processor reserves deal with allocating real time on a processor rather allocating a sequence of instructions. The reason for this is that reserving instructions would be too difficult. It would require knowing the exact sequence of instructions to be used with the reservation, fixing the exact sequence for accuracy (to avoid cache effects, etc.), allocating a time slot on the processor to execute those instructions, etc.

Several pitfalls complicate the use of this reservation model that is based on time spent on the processor. For example, DMA can impact the amount of work that gets done in a certain amount of time spent on the processor. Cache effects can introduce variance in the amount of work per time on the processor. Processor pipeline flushing at context switches decreases the amount of work done during a fixed time on the processor. These are all second order effects, but their impact should be accurately characterized.

Processor reserves leave it to the individual applications and other higher-level software to make an appropriate mapping between the computational requirements of the applications to the appropriate reservation specification. For hard real-time applications, accurately characterizing processor requirements is very important. For soft real-time applications, an adaptive approach is the key to dealing with the fact that reservations are for time on the processor rather than work done by the processor. These applications can look at their own behavior and make adjustments as necessary.

3.5.1.2 Admission control and scheduling

For processor reserves in a rate monotonic scheduling framework, a reservation request consists of three parameters: a computation time, a period, and a start time. The admission control and scheduling policies described here are based on rate monotonic scheduling [67] as described above.

3.5.1.3 Enforcement

The enforcement mechanism for processor reserves must keep track of the processor usage for each reserve so that a scheduling event can be raised at the point where the reserves allocation has been depleted for a given reservation period. The usage measure-

ment task is complicated by the fact that a thread charging to a particular reserve may be preempted, and so at each thread context switch, the usage numbers must be updated to reflect the usage since the last context switch.

Accurately accumulating resource usage

To accurately accumulate resource usage in the face of preemptive use of the resource, the system must, at each context switch, compute the usage since the last context switch. This can be achieved by recording the start time of the computation (at the last context switch) and then computing the difference between the time at the current context switch and the time of the last context switch. This is the time the last thread was using the processor resource, and this time is added into the usage accumulator for that thread's reserve. Thus the accumulator keeps an accurate account of the resource usage charged to it.

Noticing reserve depletion

The enforcement mechanism must be able to notice when the reserve of the currently executing thread becomes depleted. To do this, the system at each context switch computes the longest contiguous time the thread is entitled to execute on its reserve, and it sets a timer for that time. If a context switch occurs before the timer expires, the accumulators are updated and the timer is set for the next thread to execute. If the timer expires while the thread is executing, the system updates the accumulators, marks the reserve as "inactive", and calls on the scheduler to make some decision based on the new state of that reserve.

Replenishing a reserve's allocation

Each reserve must have its allocation replenished at the beginning of each reservation period so that the time on the resource is available if it is needed during that period. To do this, the system uses a periodic timer for each reserve which is set to expire at the beginning of that reserve's reservation period. When the timer expires, the state of the reserve is updated to reflect a full allocation of resource usage for the upcoming period.

3.5.2 Physical memory

A physical memory reserve represents a collection of physical memory pages. Physical pages are discrete resources, so they support simple discrete reservations. The more interesting question is how the owner of a page reserve uses this collection of physical pages. Basically, pages can be locked down or paged in and out, and they can be prefetched or demand paged. A small application which could fit into its page reserve would benefit from prefetching its image into the page reserve and locking down the pages. A larger application might benefit from prefetching and locking down some (more frequently used) pages while keeping other physical pages available for less frequently used logical pages to be paged in and out. The advantage of using physical page reserves for these larger applications is in the increased control reserve give the application over traditional time-sharing demand paging replacement policies. With physical page reserves, the owner of a page reserve will at least be isolated from competition for pages with other threads in the operating system.

3.5.2.1 Admission control and scheduling

The admission control policy for this discrete resource is as follows: if there are enough free physical pages to satisfy a reservation request, then the reserve is granted; otherwise the reservation request is denied. The physical page reserve contains the number of pages that are reserved, and these pages are completely free and ready to be used by the thread that owns the reserve. The system may want to keep some number of physical pages as "unreservable" pages to allow time-sharing threads enough resources to make progress.

There is no scheduling of the use of pages by the reserve mechanism.

3.5.2.2 Enforcement

The enforcement of reservations for this discrete resource is relatively straightforward: a thread that has a physical memory reserve can use pages in its own memory pool and can also use pages from the time-sharing free page pool. Thus a thread using a physical memory reserve is assured of having at least the reserved number of pages available and possibly more. At no time will the pool of pages in the reserve fall below the reserved number.

3.5.3 Network bandwidth

Reserves for network bandwidth represent reservations for time on the network device. The system must include a mechanism for identifying the reserve to be used for incoming network packets. These reserves will typically be closely coordinated with processor capacity reserves and with bandwidth reservation supported by the network. The operating system will control the amount of outgoing traffic for each session (or virtual channel), and it will ideally coordinate with a network reservation system to limit the amount of incoming traffic for each session.

3.5.3.1 Units of work

The unit of work for a network bandwidth reserve is the transfer of a number of packets of a particular size (which will probably be constant, the MTU). Servicing of single packets is certainly non-preemptive, and it should also be possible to bundle multiple packets into non-preemptive work units.

3.5.3.2 Admission control and scheduling

The reservation specification for net bandwidth reserves includes a reserved time per period of real time, and possibly an indication of expected blocking time.

Timeline or rate monotonic scheduling frameworks among others would be appropriate for net bandwidth reserves. Several important issues relate to the non-preemptive nature of the work unit. Ideally, the expected blocking time would be used in the admission control policy and scheduling algorithm.

3.5.3.3 Enforcement

Accurate measurements of usage time can be computed between requests. This information can then be used in the enforcement mechanism and for input into scheduling policy decisions.

3.6 Reserve management

3.6.1 Default reserves

Default reserves exist in the system to simplify the implementation of the reservation mechanism by providing “reserves” for non-real-time programs to charge usage against. These default reserves do not actually represent reserved resources, but they do accumulate usage for all activities that have created their own reserves or had reserves created for them.

For example, new threads are assigned to run under the default processor capacity reserve when they are created. Thus a thread will charge its time to this global reserve until it acquires a reserve of its own.

3.6.2 Composite reserves

Having many types of reserves allows flexibility in specifying resource requirements to the system and in allocating resources, but the job of managing those resource reservations at the user level becomes more involved. For example, a multimedia application, such as a video player, might reserve resources for several constituent activities. It might reserve some processor capacity for the module which reads audio and video data from the disk and passes the data to an audio server and a display server. It might also reserve processor capacity for a control module which provides fast response to interactive control commands from the user. The player part might also reserve physical memory and message queue buffers at the file system manager. Each of these reservations has an associated reserve, and we would like to be able to collect a subset of these reserves under a single name to avoid having to refer to them individually.

Grouping related reservations together helps alleviate this complexity. The model allows reservations for different types of resources, and the situation arises where a program has reserves for several different resources. Since it has to present the appropriate reserve handle to be able to use a resource, a way of grouping all of the reserves under one handle would make it easier for the program to identify its reservations to the system and to the servers it invokes.

A *composite reserve* groups reserves for different resource types under a single handle. A composite reserve has the following properties:

- it will contain a number of basic reserves,
- it may contain only basic reserves (no composite reserves),

- it may contain at most one reserve for each basic resource type

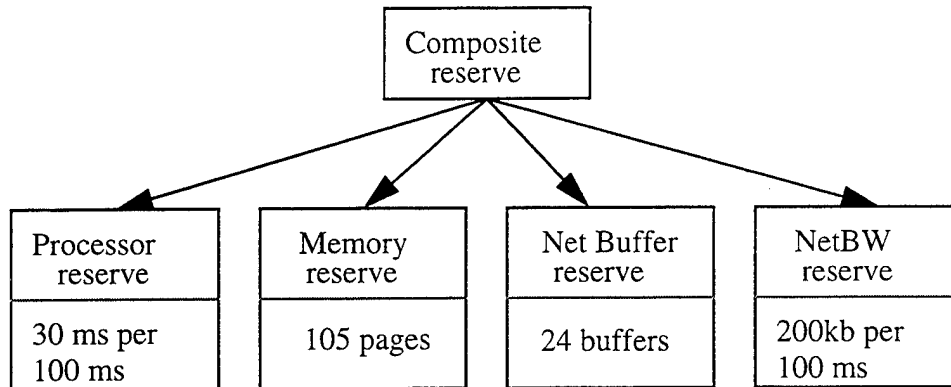


Figure 3-7: A Composite Reserve

Figure 3-7 shows the relationship between a composite reserve and its constituent basic reserves. In the video player example above, we might collect all of the reserves to be used by the player part (processor, physical memory, and message queue buffers) into a composite reserve. Then the system could use this reserve to reference the collection of resources reserved for the player. To charge computation time to the player, the system would take this reserve and look for the processor reserve under it.

3.6.3 Reserve inheritance

When a process creates a child process, the reserve of the parent is passed to the child, and the child runs against the resources reserved in the inherited reserve. This feature provides a way to allocate resources for non-real-time activities that create large process trees (like “make”). Reserve inheritance is appropriate for the automatic propagation of reserves for non-real-time programs, but real-time programs should generally configure their resources reserves explicitly. Figure 3-8 shows the difference between a process P whose children do not inherit its resource reserves and another process Q whose children and other descendants do inherit its resource reserves.

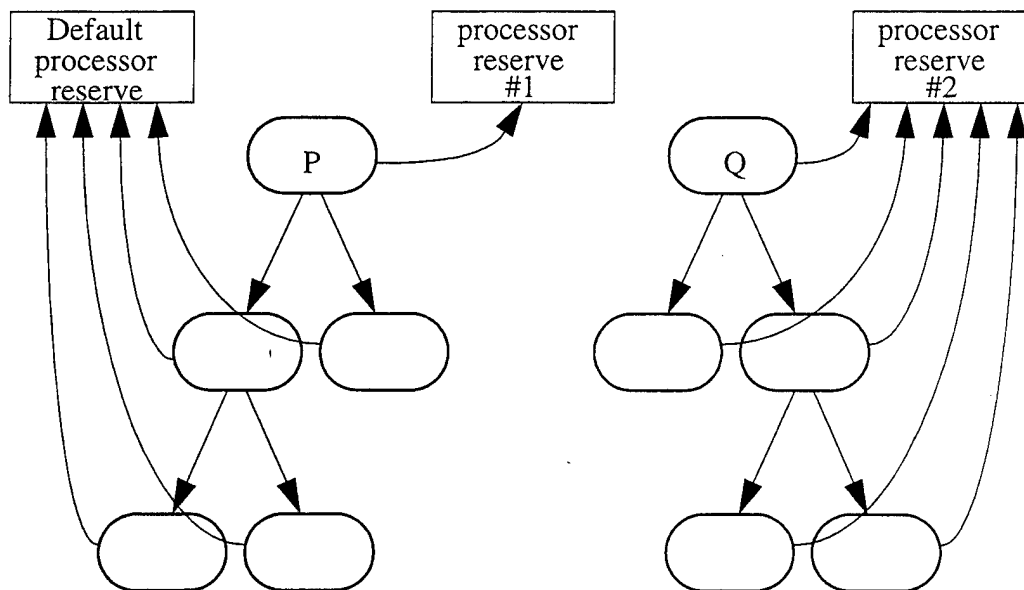


Figure 3-8: Reserve Inheritance

3.7 Chapter summary

This chapter described the basic reserve abstraction including reservation specification, admission control, scheduling, and enforcement. The idea of reserve propagation where a client hands reserves to a server to which it sends a request for service is shown to be a powerful mechanism for making reservations on a per-activity basis (rather than a per-thread basis). Several different scheduling frameworks which could be used in an operating system supporting the reserve abstraction were described, and the chapter discussed several different types of basic reserves for various resources such as: processor time, physical memory, and network bandwidth. A section on reserve management described default reserves, composite reserves, and reserve inheritance which address some practical issues in using reserves in a real system.

Chapter 4

Programming with Reserves

This chapter describes how to write programs that take advantage of resource reserves to satisfy their timing constraints. It focuses on three main issues: How should reserves be used in an application given that it uses various modules, external servers, and system services in the course of its computation? And also: How should the reservation parameters, particularly the reserved resource usage parameter, be initially chosen? How should they be adjusted given that applications must support different platforms and may have computational requirements that depend on changing input data?

4.1 Overview

This chapter describes the major issues involved in programming with reserves including the design decisions and tradeoffs that a programmer must make. The specific issues addressed are:

- How to structure programs to take advantage of reserves.
- How to map reserves onto a program's structure.
- How reservation parameters should be sized.
- How adaptive programs should adjust reservation levels.

One can think of a program as a graph of computational nodes, and each computational node has a reserve associated with it. Determining exactly what reserves should be allocated, what their reservation parameters should be, and how reserves should be associated with these computations involves design decisions that impact the program structure.

For example, the programmer must decide whether applications that depend on each other will explicitly negotiate timing requirements among themselves for the specific services they provide to each other. The alternative is to allocate resource reserves for their combined activity and then pass those reserves along as the abstract "activity" passes from

one to the other. In the first case, the partitioning of requirements and the explicit specification of timing requirements for each computation in each application creates a great deal of bookkeeping that has to be done. In the latter case, the requirements summarize the entire activity without specifying each detail along the way. As long as each phase of the activity adheres to a few rules such as not introducing unnecessary delays into the overall activity, the same high-level timing requirements can be satisfied without excessive dissection of the programs.

Another design decision addressed here relates to the flexibility of applications that use reserves. Hard real-time applications would typically specify fixed reservation parameters. Adaptive programs might be able to monitor their resource usage and adjust reservation parameters to fit their behavior over time. They might even be able to select different algorithms with different semantics and different performance characteristics to tune their computation time.

Finally, this chapter addresses the issue of programming with multiple resources. This requires applications to be broken into sub-computations at points where different resources are required. Coordinating resource reservations on multiple resources to satisfy end-to-end timing constraints requires careful design. Two approaches using reserves are described.

4.2 Using reserves in application design

This section focuses on the structure of applications and how reserves fit into that structure. Programs are considered to comprise one or more concurrent activities. Each activity might have a thread associated with it, and each activity has a call graph describing the sub-routines that are called by each subroutine. The call graph is extended to include calls to external servers or system services made by each subroutine.

The following sections describe these extended call graphs and address the coordination of reserves between reserved modules, reserved clients and servers, and reserved operating system services.

4.2.1 Program structure

To understand the timing constraints and resource requirements, one must consider the structure of application code. This section describes how an application might be divided into separate activities. It describes how a periodically executed computation in an activity might be broken down into sub-computations by splitting computations at procedure calls, remote procedure calls (RPCs), and system traps. The result is like a call graph that includes “calls” to servers and to the operating system.

For the purpose of this analysis, an activity is defined to be an abstract thread of control that starts out in a process and moves in and out of user-level servers and the operating system as calls are made to those servers and the system. This is similar to threads traversing objects in Clouds [26]. Such an abstract thread model corresponds to a synchronous programming style, which is in contrast to an asynchronous programming style where activities

are essentially “forked” by making asynchronous service requests to external servers or kernel primitives. For example, consider an animation application that synthetically generates animation frames in real time. The application consists of two activities: one to generate and display animation frames and one to process user interface events such as requests to resize the animation window.

Each of these activities may call modules in the same address space, make RPCs to servers, or make system calls. By this definition, when a (synchronous) RPC is made to a server, the activity “moves” to the server for the duration of the server’s computation and then the activity returns to the client when the call returns. If the server were to call another server synchronously, the activity would move to the second server for the duration of the call. The same is true of a system call. When a system call is made, the activity “moves” to the operating system and returns when the call returns.

An activity may be periodic. For example, consider the frame generation activity of the animation application. Suppose this activity originates in a subroutine (called `process_frame`) that is invoked periodically every 33 ms to process and display frames. Now suppose `process_frame` calls `generate_frame` and `display_frame`, which eventually performs an RPC to a window system server that accesses the frame buffer. Figure 4-1 shows an example call graph rooted at the function `process_frame`.

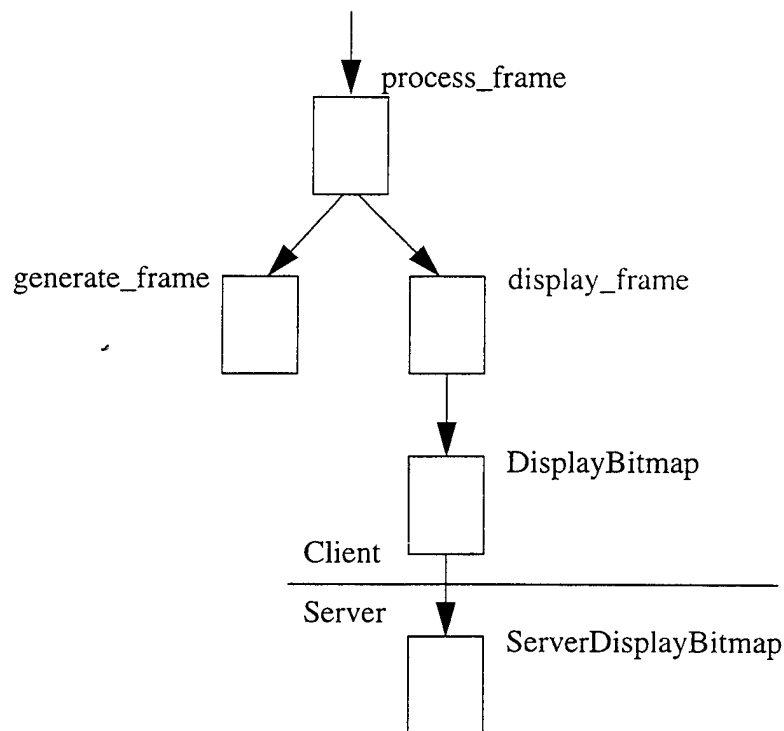


Figure 4-1: Call Graph for Frame Generation and Display

This call graph includes an RPC from the client to the server. The DisplayBitmap subroutine makes the RPC in the client, and the ServerDisplayBitmap subroutine in the server continues the activity. Thus, this graph captures all of the sub-computations of the animation activity.

4.2.2 Reservations for periodic computations

Given that the process_frame subroutine shown in Figure 4-1 is invoked periodically, the thread would “release” the computation periodically by using a while loop with a delay primitive or by setting a period attribute in the case of RT-Mach’s periodic threads [125]. Thus, the activity has an initial release time and a period parameter. To associate a processor reserve with this activity requires that a reserve be allocated with a start time and period that corresponds to that of the process_frame activity. It is possible to bind a periodic thread that attempts to execute its computation every 40 ms to a reserve that has a reservation period of 50 ms. This is not recommended, however, because the resources would not necessarily be available when the activity was released.

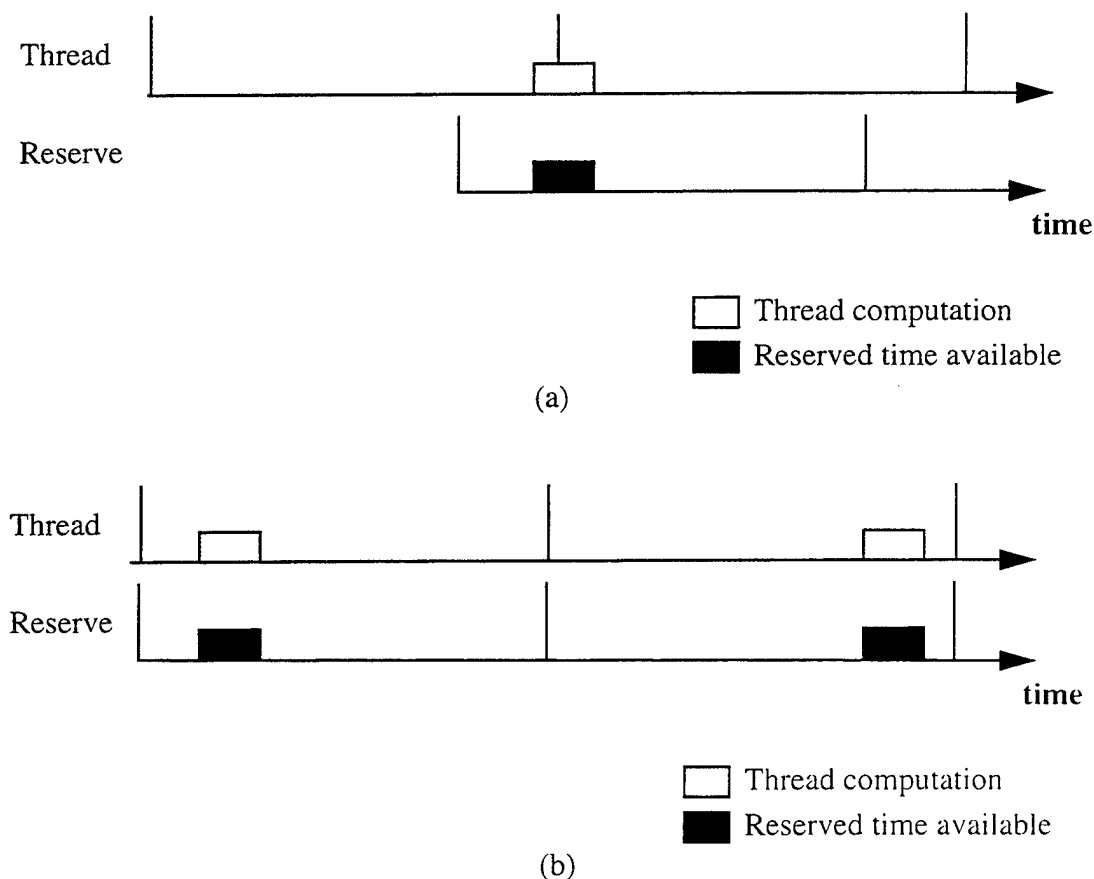


Figure 4-2: Thread and Reserve Out-of-Phase

Figure 4-2(a) illustrates a case where the thread period is not aligned with the reserve period, resulting in undesirable delays. In part (a) of the figure, the reserved computation time is not available until near the end of the thread's period. Thus the thread cannot start running until the very end of its period, and it misses its deadline at the end of the period. The problem is that the availability of the reserved time did not match the availability of the thread. Figure 4-2(b) shows the case where the thread period and reservation period are synchronized. This means that the thread will be ready when the reserved computation time is available, and the reserve guarantees that the reserved computation time will be available by the end of that period, so the thread is assured of being able to complete.

4.2.3 Localized reserve allocation

Consider the resources required in each node of the call graph in Figure 4-1. Assume that `generate_frame` requires only the processor. For nodes under `display_frame`, assume the frame buffer is mapped into the window system server's address space and that the processor is the only resource required.

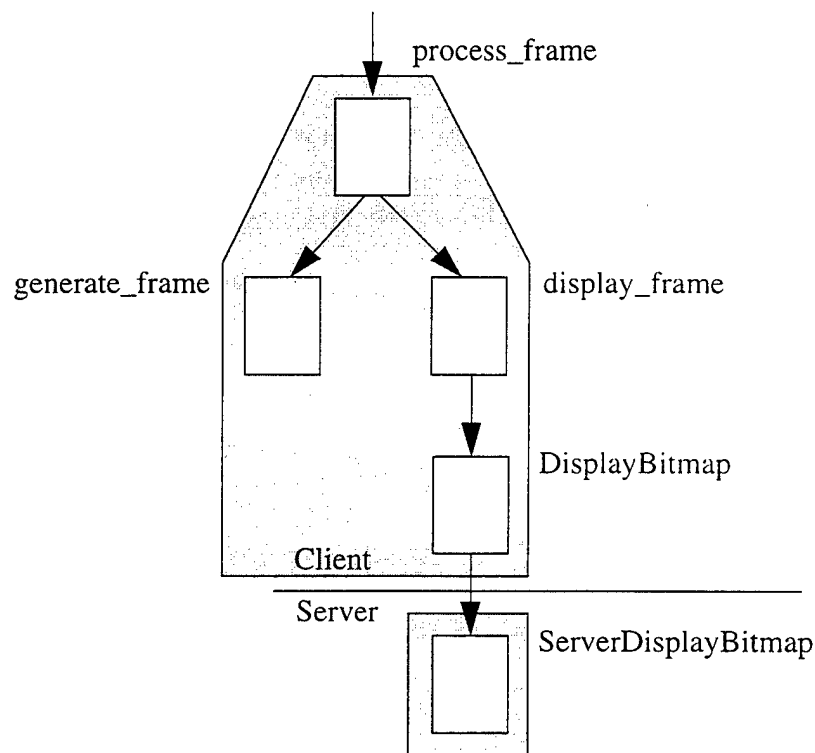


Figure 4-3: Call Graph with Separate Client and Server Reserves

With these assumptions, one approach to allocating reserves for the sub-computations would be to allocate a processor reserve for all of the nodes in the animation application and another processor reserve for the nodes in the window system server. Thus, the server would have a reserve allocated for each of the clients holding open connections to it. This approach

is necessary in the case where the server resides on a remote machine, but it may be preferred even when both utilize the same processor. Figure 4-3 illustrates this approach.

The RPC from the client to the server implies a switch from the client's reserve to the server's reserve. With this approach, the traditional "priority" inheritance mechanism would not be useful because the fact that the server has a reserve allocated internally for the animation client defines the "priority" for the client's request in the server and the client's "priority" does not get propagated. Another mechanism to associate the animation applications RPC request with the appropriate reserve inside the server would be very useful. Such a mechanism might take the "priority" of the server-allocated reserve to be associated with the animation client and apply it to the thread that will handle the client's request. This is a kind of "priority" inheritance where the server's thread gets the priority of the reserve it allocated for a client instead of getting the priority of the client itself.

Since the server must allocate the reserve for its computation on behalf of a client, it must know what the reservation parameters should be for the reserve. This approach requires the client and server to enter into a dialogue to allow the client to explicitly request a server-specific QOS level, meaning a certain pattern of server operations to be called with certain timing constraints. The server must then map the requested QOS requirements to system resource requirements and decide whether it can acquire the reserves to support that activity. All of this negotiation must be explicit, and that means a client/server interface for negotiating server-specific QOS requirements must exist. Further, the server must have the machinery to map those QOS requirements to system resource requirements.

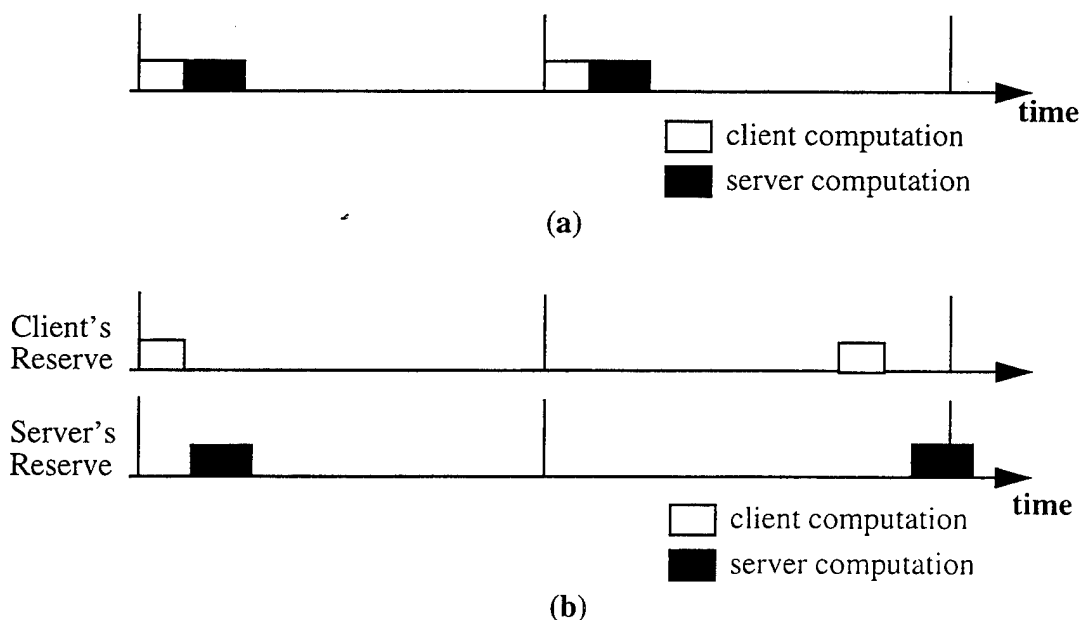


Figure 4-4: Switching Reserve from Client to Server

Another issue is that the timing parameters of the two reserves must be carefully coordinated for the call to be executed smoothly. Essentially, the client's call to the server means that the computation in the server becomes ready, and its reserve must provide the resources for it to execute in a timely fashion after it is ready. The sequence of client computation followed by server computation is illustrated in Figure 4-4(a).

The server's computation time might be available immediately after the call is made, as in the first period of execution history shown in Figure 4-4(b), in which case the deadline for the combined activity is met. But as shown in the second period of the execution history in Figure 4-4(b), the client's computation time may be available very late in its period. The server's computation time may be available earlier in its period but not available so close to the end. It is guaranteed to be available sometime in the period, but not at any particular time. Thus the activity could miss its deadline.

Introducing an intermediate deadline for the client's computation could solve this synchronization problem. Figure 4-5(a) shows the usage pattern of the client and server with an intermediate deadline for the client.

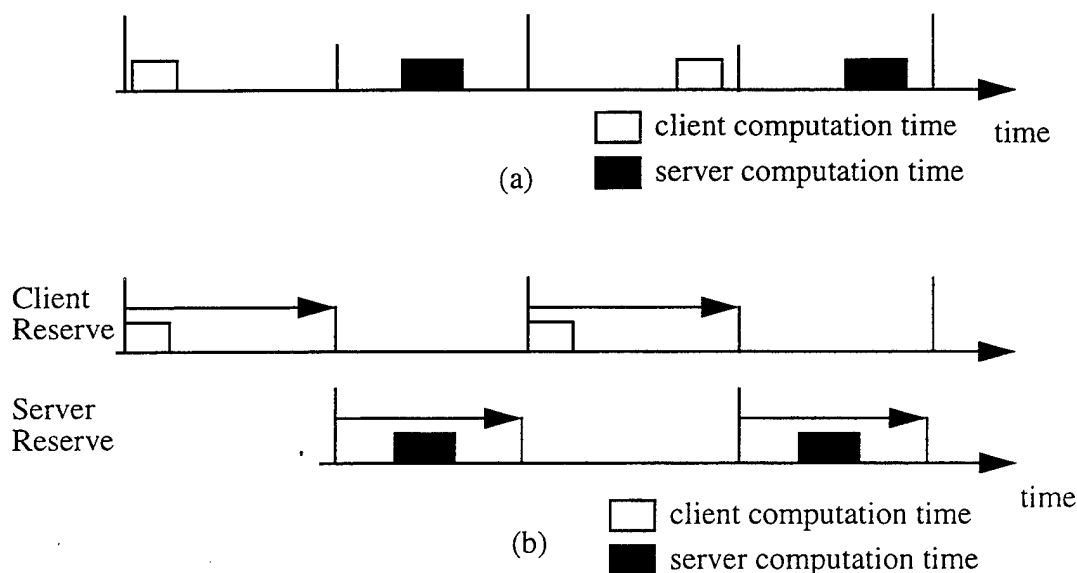


Figure 4-5: Client Requirement with Intermediate Deadline

Using reserves with deadline parameters, Figure 4-5(b) shows how the reserves can be allocated such that the client reserve has its computation time available at the beginning of the client's period with an intermediate deadline halfway through the period. The server's start time is at that intermediate deadline, and it has a deadline that corresponds to the end of the client's overall period. Thus, both activities are guaranteed to synchronize and complete by the overall deadline as desired.

All of the explicit handling of QOS requirements and resource requirements and the careful synchronization of interactions between reserves makes programming clients and servers much more complex. While this may be necessary for designing complex hard real-time systems, for soft real-time systems and less complex hard real-time systems, the approach where resource requirements for clients and servers are folded into one reserve may be better.

4.2.4 Activity-based reserve allocation

Another approach would be to allocate a single processor reserve for all of the nodes in the entire call graph; Figure 4-6 illustrates this approach. Of course, if the server is running on a remote host, this approach may not be feasible since it is not clear how a single processor reserve could be made to represent processor time on two different processors.

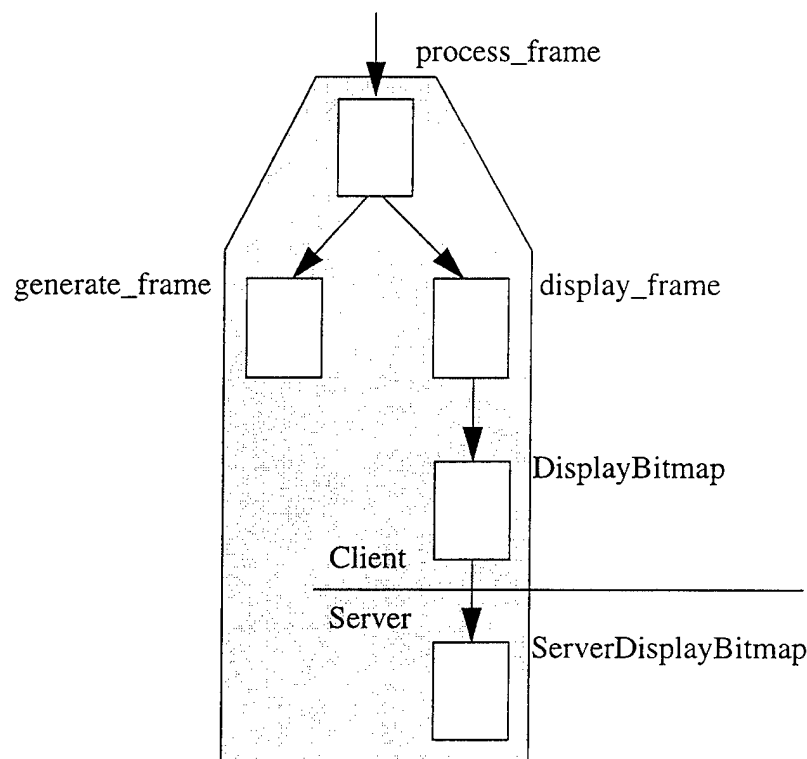


Figure 4-6: Call Graph with One Reserve for All Nodes

Even if the animation application and the server are on the same host running on the same processor, there is a problem that must be addressed in this approach: the server may not be ready to service the RPC call at the time it is issued. In fact, there is a potential “priority” inversion problem associated with such an RPC (where “priority” refers to the ordering of reserved activities by the scheduler rather than an integer priority for a thread). If the

RPC arrives in the server's queue at a time when the server is servicing another client, the server's thread will be bound to the reserve associated with the other client. If that other client is processor-poor, the reserved time may run out during the service, and the server may experience some scheduling delay. If the other client has a reserve that happens to get its processor time very late in its period, there may be a significant delay until the server, running with that other client's reserve, can finish the on-going operation.

To limit the delay the animation application experiences waiting for the server to handle its RPC, a "priority" inheritance protocol [108] must be employed. If the animation client's reserve would be sequenced by the reserve scheduler before the other client's reserve, the server which is using that other client's reserve would be sequenced as if it were using the animation client's reserve. However, for consistency of the usage measurement, it will still charge its usage to the other client's reserve. Then when that service is finished, the server will bind to the reserve of the animation server and that completes the propagation of the animation application's reserve to the server.

For this reserve passing to work best, the RPC should be synchronous, meaning that the client should wait for the result after making the call to the server. With a synchronous RPC, either the client or the server will be charging against the client's reserve whereas with an asynchronous RPC where the client does not wait for the result from the server, both the client and server may be charging against a single reserve at the same time. This is not catastrophic, but it may result in complicated interactions between the client and server.

This approach implies that the client application must request reservation parameters that include the computation time that will be consumed by the nodes residing in the server. This can be done by having the client discover the requirements empirically during runtime, by having the server explicitly provide its resource requirements, or by determining the requirements at design time (this issue is discussed in detail in the next section).

The important point here is that the client and server need not explicitly exchange information about resource requirements if the client allocates the reserve and passes it to the server. In particular, a great deal of complexity can be avoided if the client/server interface does not need to be able to support a complex negotiation of requirements. For legacy systems, this means that existing interfaces need not be radically modified, the only change being the mechanism for passing reserves from client to server.

4.2.5 Coordinating multiple resources

This section describes an issue that arises when an application uses multiple different kinds of resources in different sub-computations. Consider an audio/video player application that reads data stored on a disk and then outputs an audio stream and displays video frames. The player could be structured as three activities: audio playback, video playback, and user interface. The video playback activity would be periodic, reading and displaying a frame every 30 ms.

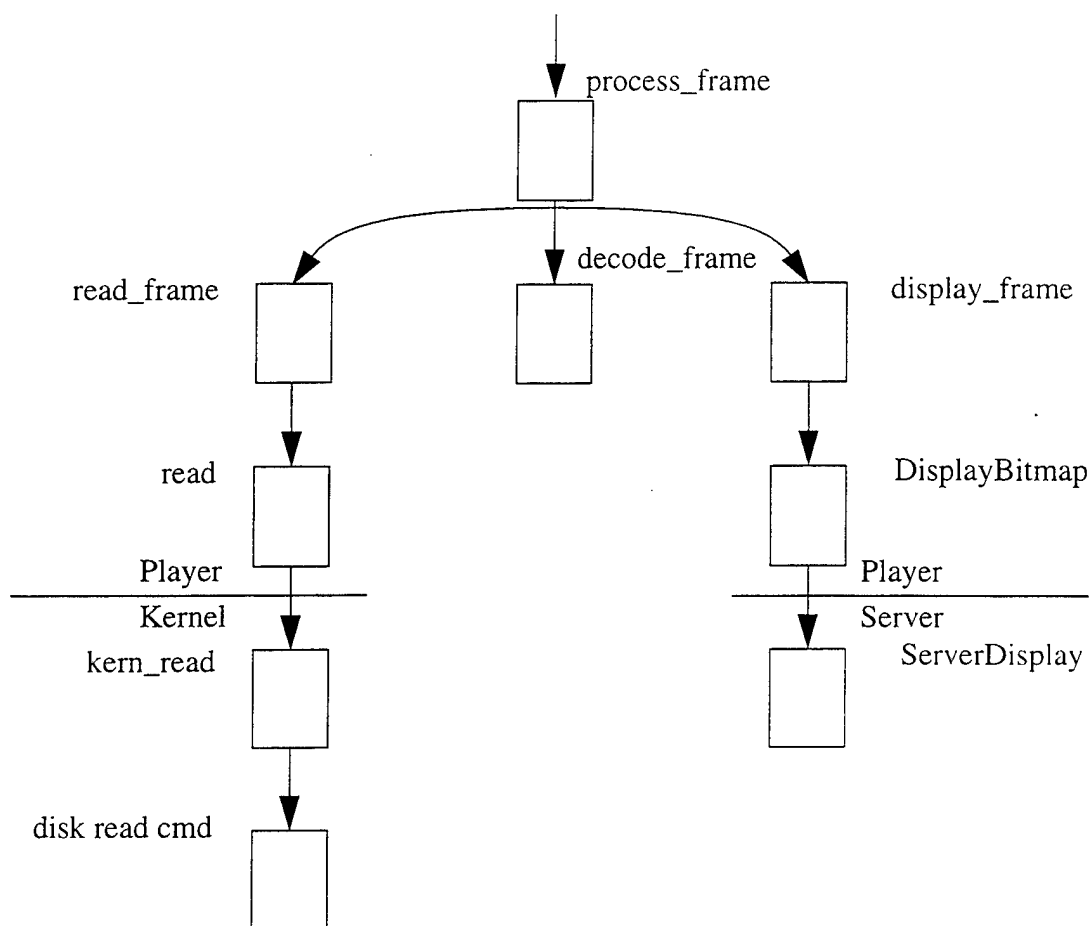


Figure 4-7: Call Graph for Video Playback

Figure 4-7 shows a possible call graph for the video playback activity. The graph is rooted at a subroutine called `process_frame`, which calls three more subroutines: `read_frame`, `decode_frame`, and finally `display_frame`. In the `read_frame` routine, calls are made until eventually the program makes a system call and traps into the kernel where more subroutines are called until finally a device command is issued to read data from the disk. This call graph introduces another type of call, referred to as a device command, which is used in addition to the original three types of calls (procedure call, RPC, and system call). The `decode_frame` routine converts the video frame data to a form that is suitable for display. The last call is to `display_frame` which is the root for a sequence of calls resulting in an RPC to a window system server which makes additional subroutine calls and finally accesses the frame buffer.

The resource requirements for this call graph include disk access as well as processor time, so a disk reserve is allocated and bound to the disk read node. The other nodes require

only the processor, and one approach is to allocate a single processor reserve for all of those. This reserve allocation and binding approach is illustrated in Figure 4-8

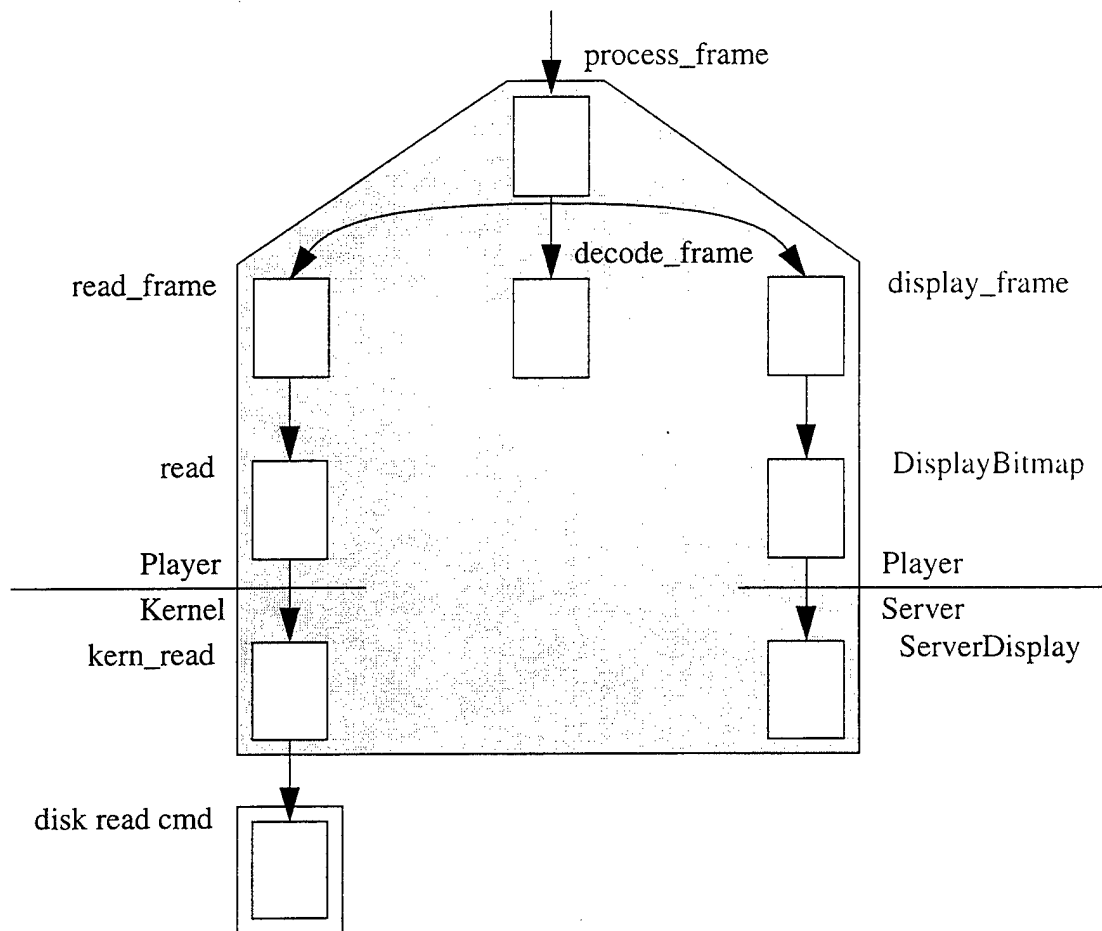


Figure 4-8: Call Graph for Video Playback with Reserves

There is a subtle problem, however, that is related to the synchronization problem between and client and server with localized reserve allocation. The resource usage pattern for the `process_frame` activity is the following:

- The processor is needed for all the nodes up to where the read command is issued to the disk,
- the disk is required for that read command node,
- all the nodes after that require only the processor.

This resource usage pattern is illustrated in Figure 4-9(a); in every period, the computation first has a processor requirement, then a disk requirement, and then another processor

requirement. If a processor reserve and a disk reserve are allocated, the execution pattern may look like the pattern shown in the first period of Figure 4-9(b).

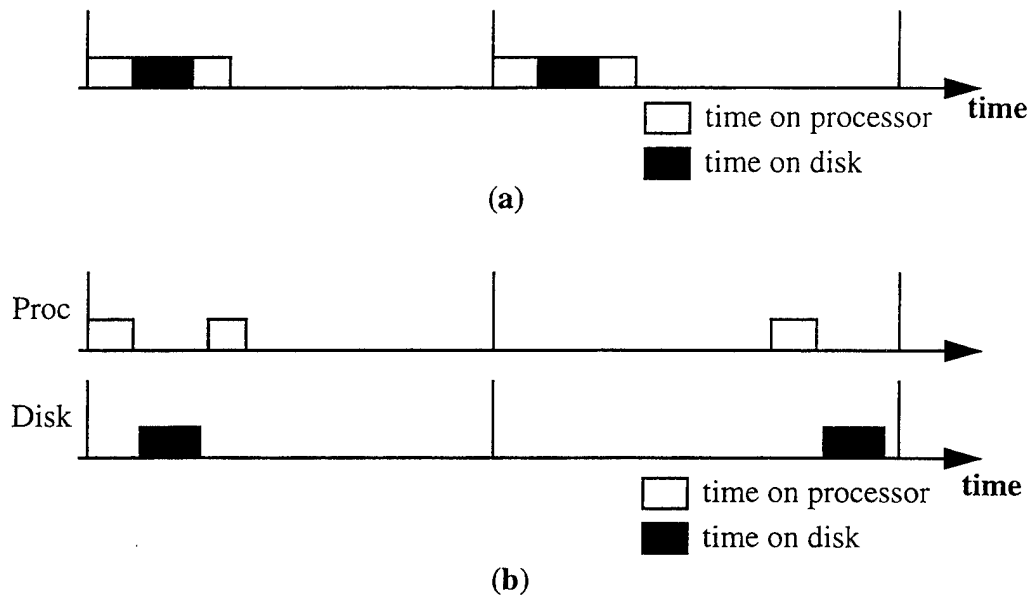


Figure 4-9: Synchronization Problem with Multiple Resources

However, the only guarantee associated with the processor reserve is that the processor time reserved will be available by the end of the period. If that time happens to only be available at the very end of the period, the execution pattern might look like the one in the second 33 ms period of Figure 4-9(b). In that second period, the leading processor requirement is serviced too late, and by the time the disk activity is finished, there is no more time left in the period for the second half of the processor requirement, and the deadline is missed. Worse still, if the reserved disk usage is only available at the beginning of the disk reservation period, the activity will be delayed into the next reservation period and certainly miss a deadline and possibly miss the following deadline as well.

One way to solve this problem is to introduce intermediate deadlines in different stages of the computation to separate sub-computations that use different kinds of resources. For example, Figure 4-10(a) illustrates the usage requirements and new intermediate deadlines for the video playback activity.

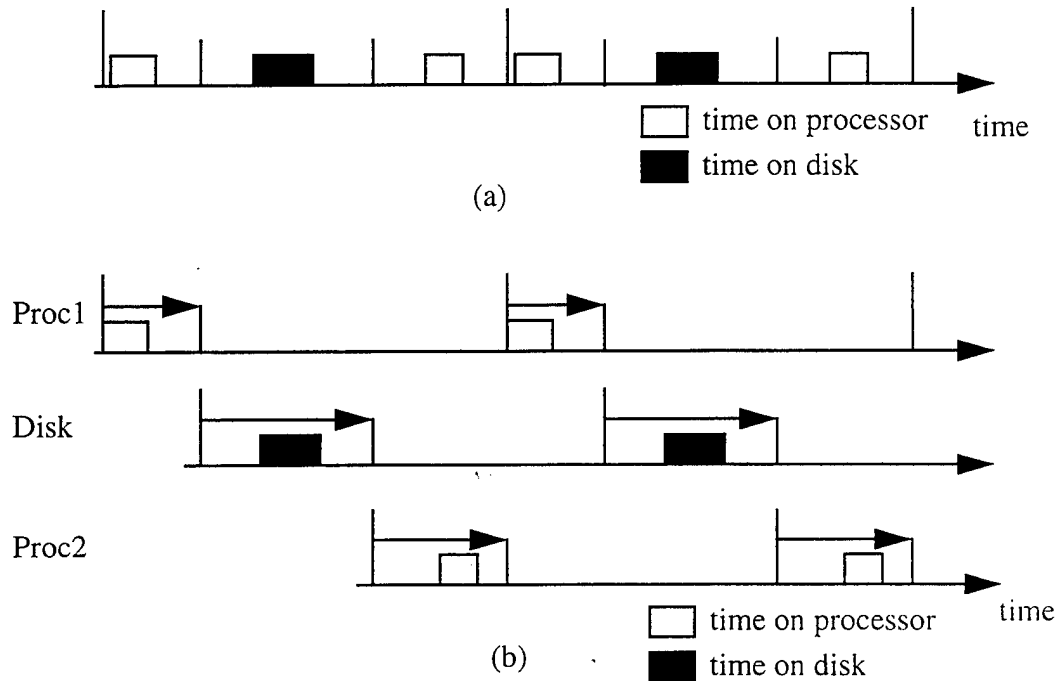


Figure 4-10: Multiple Resources Used with Intermediate Deadlines

A reserve is allocated for each of the three phases of the computation: the leading processor requirements, the disk requirement, and the final processor requirement. Since the end-to-end timing requirement or deadline is divided up into intermediate deadlines for performing the three phases of the overall computation, the reserves that are associated with the phases must have deadline parameters. Figure 4-10(b) shows a timeline for each reservation and how the usage is timed in the three reserves.

So with this approach, two processor reserves (labeled Proc1 and Proc2 in Figure 4-10) and one disk reserve are allocated. The call graph with this reserve allocation and binding appears in Figure 4-11.

This example points out two major factors that influence how reserved computations should be structured and how reserves should be bound to the sub-computations. One factor is the temporal sequence of the resource requirements. Generally speaking, a node in the graph that requires a resource different from its parent acts as a delimiter for grouping computations that can use the same reserve. To minimize the number of reserves required, the application programmer should minimize the number of times computations must switch between required resources.

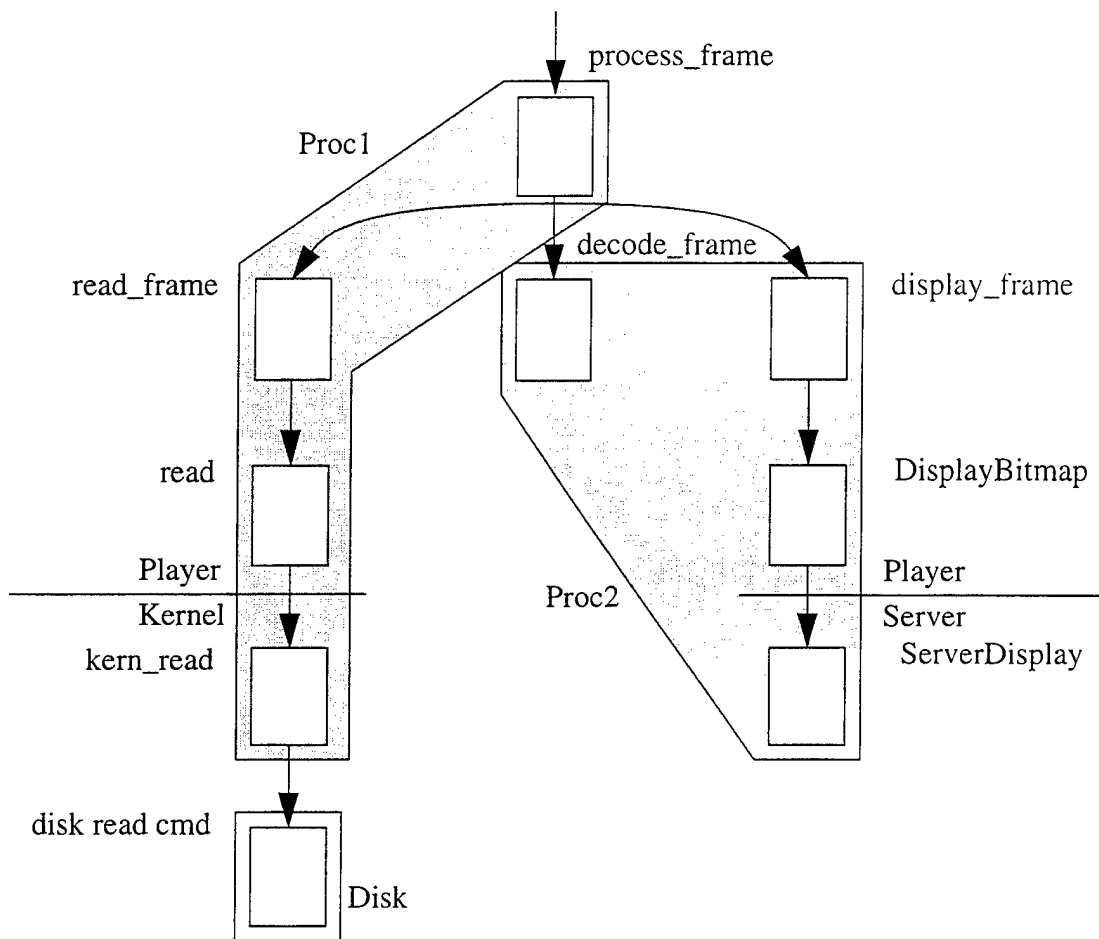


Figure 4-11: Video Playback with Better Reserve/Computation Mapping

The second factor is the spatial organization of the nodes in the system. When a call crosses to another address space or to the operating system, a decision has to be made about whether to switch reserves at that point or not. The system designer has more control over that choice, as described above.

4.3 Sizing Reservations

To use resource reserves, an application must specify appropriate reservation parameters. For hard real-time applications, the reservation parameters would be determined *a priori* by the system designer. For dynamic real-time applications, external agents such as a QOS Manager may suggest or require different reservation parameters during the course of the application's execution. In the dynamic framework, it is important for the application to be aware of the resources required to do the work it needs to do and to be flexible in terms of

its timing requirements (how often or under what delay bounds it does the work). In general, two important questions must be addressed:

- How can the resource usage requirements of an application be determined, especially given that the application may be used on different machine types and system configurations?
- How should an application adjust its resource reservations using information about previous performance?

This section deals with the determination of initial resource reservations and adjustment of reservation levels for dynamic periodic activities.

4.3.1 Determining resources required

The first problem to be addressed is how to determine which resources will be needed during the course of the computation and how to determine the initial reservation levels for the various resources required by an activity.

The resources that are needed during the course of a computation will depend on what external services are used by the computation. The list of resources for the computation will be the union of all the resource lists for the transitive closure of external services used by the computation. It is therefore very important for services to be named so that each service can name those services that it uses. And in turn, each service must name those resources that it uses. Then it is possible to find the resources used by the transitive closure of services the application uses.

A potential problem is that different functions offered by a service may use different resources. If a client uses only one function offered by the service, it should reserve only the resources needed for that function rather than the complete list of resources needed by every function the service offers. In this case, it may be useful to consider the resource lists required for “sub-services” or subsets of operations of the service where the subsets are defined to use similar sets of resources.

In any case, the method for ascertaining required resources should be flexible, efficient, and easy to use. Ideally, the system would help to determine the list of resources during an initialization phase of each application. Each time the system encountered an application that required a particular resource but had no corresponding reserve, it would add a reserve of the appropriate type with no reservation parameters to the reserve tree bound to the application. After the initialization phase, the application would have references to the resource that were required by its component computations, even those resources used by servers that were called on its behalf.

Other approaches to determining resources required could be used as well including the following.

- The list of resources could be obtained by sending a query to every server to be used in the computation and having the server provide a list of resources it requires and a list of services it uses. The transitive closure of

required resources collected during this query would be accurate as of the server connection time. This method does not require that the system and (possibly more static) documentation remain synchronized with respect to specification of resources used. Upgrades for various system software modules could be made without having to issue new resource list documentation. Also, servers would be free to determine which resources, among many possibilities, would be best to serve that connection given the state of the system and its load at the time the connection was requested. Thus an additional degree of freedom is allowed the servers.

- The list could be found using a database where each server registers the list of resources as well as other services it requires. This makes it possible to write applications that automatically determine the transitive closure of services used and resources required, even if some of those services and resource types did not even exist at the time the application was developed and compiled. One important requirement to make this dynamic method work is that service names and resource types not be hard-coded. Instead, a program should be able to handle and manipulate new service names and resource names with no recompilation.
- The list of resources required by various user-provided services and system services could be static, long-lived, and well documented. The programmer must manually look up all the services and find the transitive closure of services used and then the union of the resource lists of all those services. The major problem with this approach is that the slightest changes to the software for the services may change the list of services used and the resource list, thus making the lists in the manual obsolete and making all the programs written to the specification of the manual obsolete.

4.3.2 Determining initial reservation levels

Once the programmer knows what resources are needed by an activity, she must set up the reserves for those resources and request reservations. Requesting reservations requires that reservation parameters be provided. In the reserve model, a reservation request has parameters for resource time to be reserved and for a reservation period. In many cases, the reservation period will be the same as the period of the activity. This will sometimes be derived directly from user-level quality of service requirements (such as frame rate), and sometimes it will be derived indirectly from user-level requirements. For example, the rate for handling audio packets might depend on the audio sampling frequency, the packet size, and perhaps the system overhead per packet. The resource usage time is more difficult to ascertain. It depends on the platform, the system software, and the data being processed among other things.

One way to get a reasonable estimate as to what the resource usage requirements might be for a given instantiation of an application involves measuring the actual computation that forms the main focus of the application. With one run through a periodic activity, for exam-

ple, the application could get a fairly good estimate of future computation times using the reservation mechanism's usage measurement features. Another variation on this approach is to use a simple computation to gauge the speed of the machine and/or system architecture, and then use a characterization of the real application's computation expressed in terms of the simple computation to estimate the appropriate reservation level. For example, if the application first ran a SPECint benchmark and knew how much the reserved computation needed in terms of SPECint benchmarks, it could derive the estimate directly.

The following methods could also be used to determine the initial reservation level:

- An application could store in a persistent preferences database some information about reservation levels used in previous instantiations of the application. This information would be a good guess as to what reservation levels should be procured, and it might be possible to maintain a small database to map prior experience with different QOS parameters to reservation parameters. This approach might get much more complicated as more QOS parameters, reservation parameters, and target system architectures are used.
- The initial reservation level could be set to zero or some other relatively small value that is known to be smaller than the actual reservation level, though unknown, that will be required. This approach requires the mechanisms for reservation level adaptation to quickly acquire the feedback on usage that is necessary to set a reasonable reservation level where desired quality of service parameters can be achieved. Initially, the desired QOS parameters will not be achieved and they may never be achieved. These are the major drawbacks of this no-knowledge approach.
- An alternative approach to the zero level initial reservation is to take the maximum reservation level available on the resources at the time the reservation is requested. This has the advantage of having the highest chance of meeting the desired QOS parameters for the application, but the disadvantage is that resource capacity may be unnecessarily tied up and unavailable to other applications requesting reservations. This situation would persist until the adaptation mechanism had the chance to evaluate the situation and make the proper adjustments to the reservation levels.

4.3.3 Measuring performance

An adaptive reserved application should keep track of the resource usage required to perform its computation at each repetition to decide if it has more resource capacity reserved than it needs or if it has too little resource capacity to do its work during each period. It should also keep track of the real-time delay incurred during each repetition of the computation to determine whether the computation was completed within the period or not.

The application can easily measure the real-time delay of a computation by taking a timestamp at the beginning and at the end of the computation. Measuring the resource

capacity usage for a computation, however, is more involved, requiring support from the operating system. This support must be more accurate than the traditional logical clock for processor time provided to processes in most operating systems. Logical clocks usually take usage measurements by sampling at clock interrupts to find which process is running and incrementing that process's logical clock as if it had been executing for the entire period. Statistical sampling of this kind, which is inherently inaccurate for short-term measurements, will not provide an application with the clear picture of short-term behavior. Such knowledge of short-term behavior is needed to be able to make suitable adjustments to the reservation parameters.

For the kind of accuracy required for measurements of resource capacity usage, the system must accumulate usage associated with reserves at each context switch. In this context, reserves act as *abstract thread* logical clocks rather than *process* logical clocks. And since the reservation system manages capacity usage for resources other than just the processor, the system must keep usage accumulators for all types of resources, and these must be updated at each context switch on the appropriate resource.

Reserve usage measurements will indicate how an application's actual behavior is related to its reservation. Several possible patterns of behavior are described in the next sections.

4.3.3.1 Balanced applications

An application is balanced with respect to its reservation if the resource usage in each period is fairly constant and the reservation level is at this constant value. (It may be impossible for resource usage to be completely constant for some interesting resources such as processors.)

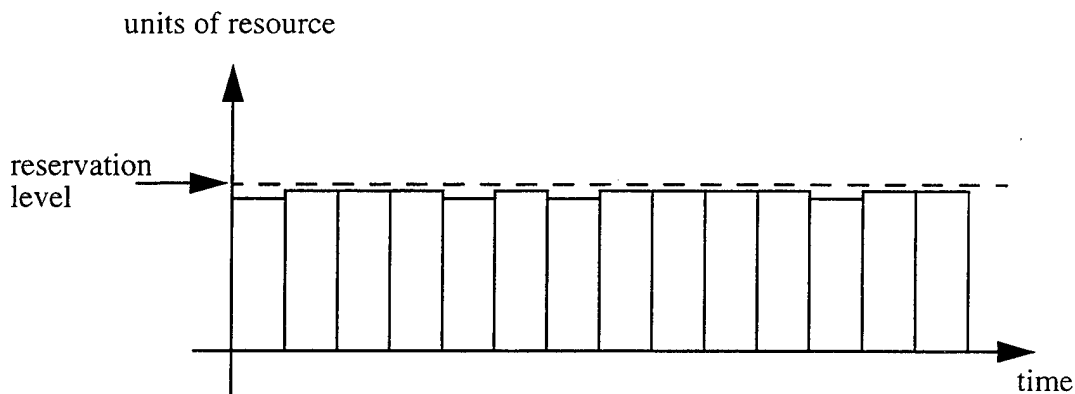


Figure 4-12: Resource Demand Constant and Reserved

Figure 4-12 illustrates a computation's demand on a particular resource over time. Time is on the x-axis, and it is divided into intervals equal to the reservation period. The y-axis is units of resource, e.g. time spent on the processor executing instructions, bytes transmitted,

etc. Within each reservation period, the number of units of resource consumed by the computation is measured, and the height of the bar in that interval is the number of units consumed.

In the figure, the number of units of resource required in each reservation period is nearly constant, and the reservation level is slightly more than this constant demand. Therefore, the demand is satisfied by the reservation, and the computation will have the resources to be able to execute completely in each period.

4.3.3.2 Under-reserved applications

An application is under-reserved with regard to a particular resource if its resource usage requirement is greater than its reservation. Two cases are distinguished:

1. worst-case (maximum) resource usage requirement for the computation is greater than the reservation but the average resource usage requirement is less than the reservation, and
2. the average resource usage requirement is greater than the reservation (implying that the worst-case resource usage requirement is also greater than the reservation).

In the first case, the average resource usage requirement is less than the reservation, so over the long term, the application will be able to keep up with its work requirement. The problem is that since the worst-case resource usage requirement is larger than the reservation, the completion of the worst-case computation may be delayed and this may delay or otherwise affect the computations in subsequent periods. If the worst-case computation occurs very infrequently, its negative affects on the overall performance of the application can be minimized or ignored. A human viewer may not even notice an occasional dropped frame during video playback. If the worst-case computation occurs frequently, it may be more difficult to ignore; many dropped video frames would certainly be noticed.

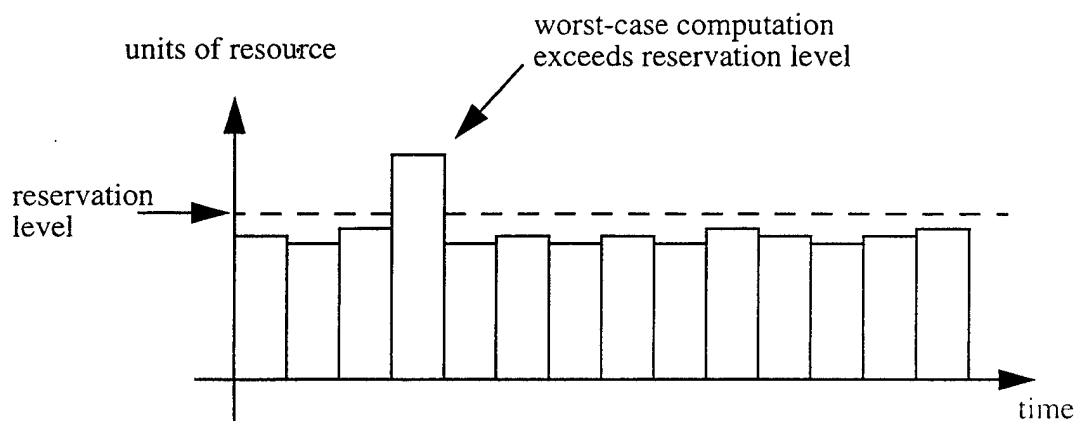


Figure 4-13: Resource Demand Occasionally Exceeds Reservation

Figure 4-13 illustrates this slightly underreserved case. The resource usage requirement in most reservation periods is less than the reservation level for this particular resource. In these periods, the computation will have the resources available to complete. However, there is one period in the illustration (the 4th) in which the resource usage requirement is larger than the reservation. Depending on the system's policy for treating this case, the computation may happen to be completed (using idle time), it may be aborted, or it may extend into the next reservation period, interfering with the completion of the computation which would normally execute in that period.

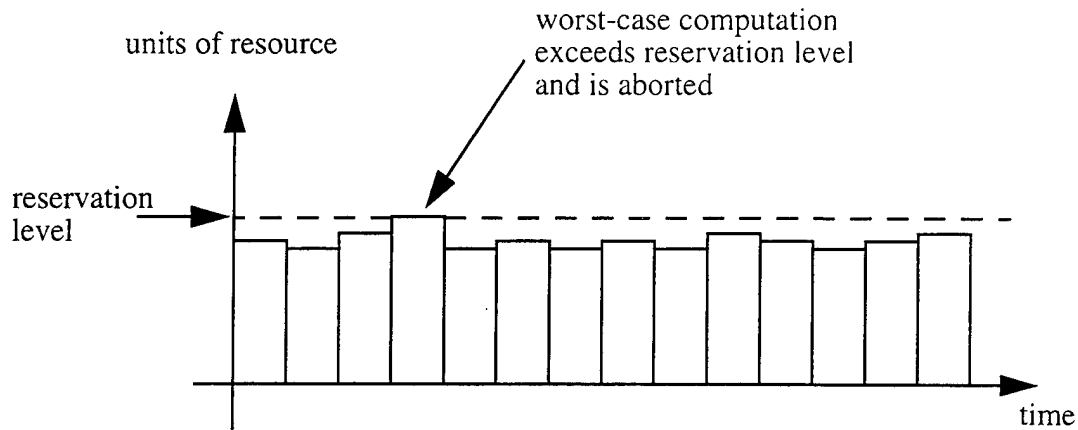


Figure 4-14: Exceedingly Demanding Computation Aborted

Figure 4-14 shows how the usage pattern would look if the computation in the 4th reservation period were aborted. Note that the computations in the subsequent periods are not affected by the aborted computation.

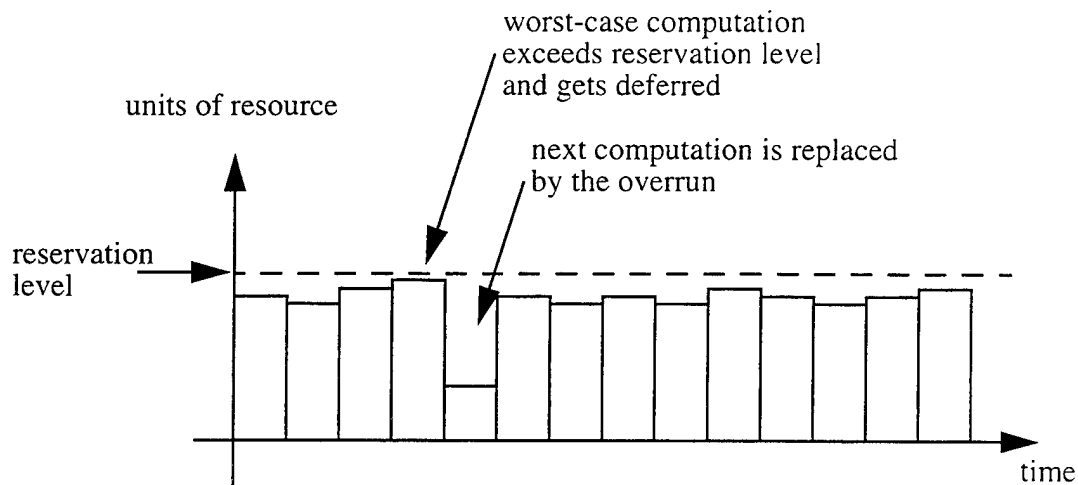


Figure 4-15: Computation Impinges on Following Computation

Figure 4-15 shows the case where the computation in the 4th reservation period extends into the next reservation period and prevents the next computation from being initiated. Computations following that are left undisturbed.

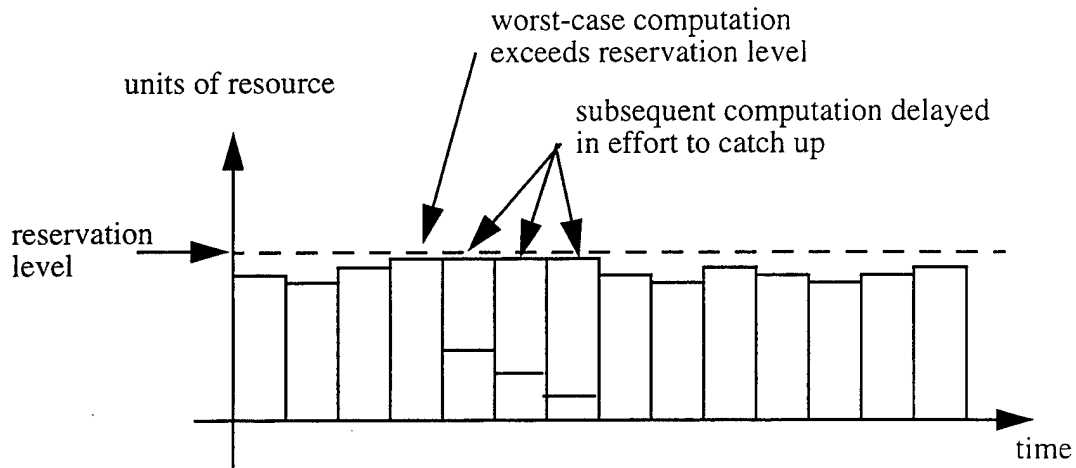


Figure 4-16: Computation Impinges on Subsequent Computations

In Figure 4-16 the 4th computation overruns its reservation period and consumes part of the next period. The computation associated with that period is initiated after the previous computation is completed (as opposed to the previous case where this computation was not initiated). But since the 5th computation is initiated later than usual, it also overruns its reservation period and is deferred to the next period. This cascading effect continues until there is enough (normally) unused but reserved units of resource to make up for the original overrun.

In cases where the average resource usage requirement is more than the reservation, the activity will never be able to accommodate all of the computations which overrun, and it would be necessary to shed some of the load by aborting some computations or by not initiating some computations. In either case, the attempted overruns would occur frequently and have a potentially damaging effect on overall application behavior. In a video player, for example, this would mean that many frames get dropped.

Figure 4-17 illustrates a possible pattern of demand that has the average demand greater than the reservation level. The computations in several reservation periods require more than the reservation for that period. Many of the computations will have to be aborted if there is no idle time available beyond the reserved level.

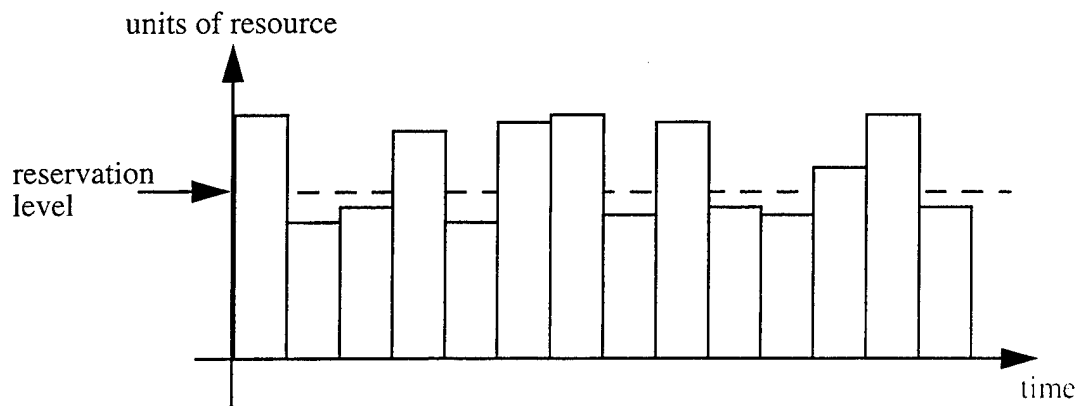


Figure 4-17: Average Demand Exceeds Reservation

4.3.3.3 Over-reserved applications

An application is over-reserved if the resource usage in each period is (much) smaller than the reservation level.

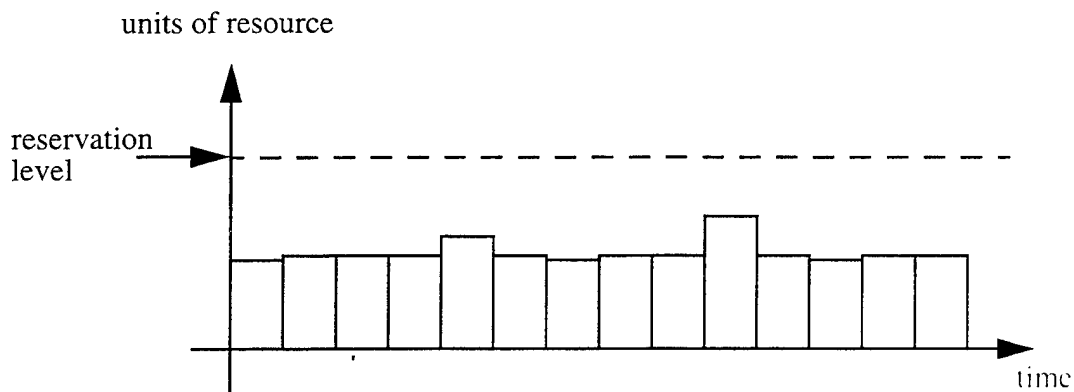


Figure 4-18: Resource Demand Smaller than Reservation

Figure 4-18 illustrates a case where the usage on a particular resource is much smaller than the reservation in all of the reservation periods. Here the computation is never in any danger of overrunning into the next period.

4.3.3.4 Multiple resources

When there are multiple resources involved in each computation, the measurements of usage compared to reservation level for each reserve will be different. One resource may be over-reserved while all of the others are under-reserved, or perhaps more commonly, one

resource may be under-reserved (representing a bottleneck) while all of the other resources are over-reserved for the activity.

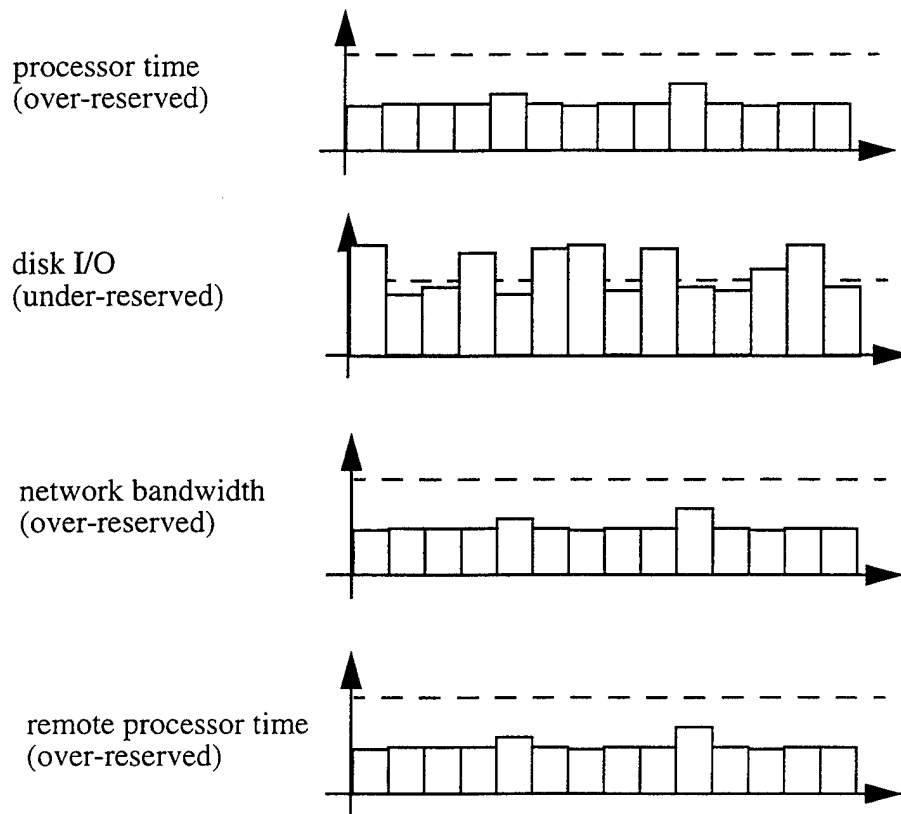


Figure 4-19: Measurements of Multiple Resources

Figure 4-19 illustrates a case where there are multiple resources involved in a single activity. They all have the same reservation period, but the demands placed on various resources are different. In this case, the disk I/O is under-reserved while the local processor usage, the network bandwidth, and the remote processor usage are all over-reserved.

Once an application has measurements of usage for the various resources it requires for its computation, it can begin to make decisions about how to modify its own behavior or modify its own resource reservation levels to achieve better performance or better efficiency.

4.3.4 Adapting

Reservation parameters can be changed dynamically as the user, the application itself, or a central quality of service manager determines that new reservation parameters would be preferable. Applications adapt based on the influences of various external entities, but once a resource reservation is made, the system ensures that the resources are available.

Adaptive applications can measure their own performance by mapping measured system resource performance metrics to application-specific performance indicators. This application-specific information can then be used along with application-specific performance objectives to:

1. modify the computation being done by the application (to change resource requirements),
2. modify the reservation level for resources being used by the application, or
3. do nothing.

The performance measurement interval could be comparable to the period of the repetitive computation, but it would be more efficient if the adaptation interval were an order of magnitude larger than the activity period (e.g. the adaptation might occur every 500 ms for an activity with a 50 ms period).

4.3.4.1 Modifying an application's computation

One way that an application might react to the fact that its resource usage is different from its reservation is to change its behavior so that its usage more closely matches its reservation (leaving the reservation unchanged). The actual mechanisms for modifying behavior in an application are fairly straightforward. An application which is meant to modify its own behavior must have different behaviors available (i.e. different algorithms implemented internally). It must be able to tell which algorithm it should use depending on the format of an incoming or outgoing data stream, on the resources such as network bandwidth or computational power that are available to it, or on the limitations of other software with which it must interact. For example, an MPEG video decoder could use different decoding or dithering algorithms depending on the resources available. If the decoder were taking the MPEG data stream from a server in real time, it might be able to negotiate MPEG encoding parameters with the server and have the server place new parameters in the data stream. Thus, adaptive applications are constrained by:

1. the algorithms they have coded,
2. the data formats they are using,
3. the data formats and data rates that other components of the pipeline can handle.
and
4. the resources that are available to it.

Typically, the application would contain a collection of algorithms that could be ordered based on processor requirements, network bandwidth requirements, etc. Thus, once an adaptive application decided to increase or decrease a reservation on a particular resource, it could determine which algorithms could satisfy that constraint. It is important to distinguish between two kinds of behavioral adaptations:

1. local changes in algorithm and
2. global changes.

An example of a local change would be a change in the number of bits actually being decoded by the receiver of a bit stream (assuming the bit stream was encoded using a hierarchical encoding scheme). A global change would involve not only the receiver but the sender. It would require a way for the receiver to request a change in the format or number of bits being sent as well as requiring the receiver to recognize that the format of the bit stream changed.

As an example of what the code structure for an adaptive application would be, consider a video player. The basic control structure of the player is a loop that reads some data (from a disk, network or some other source), decodes the data, displays the data, and evaluates its performance.

```
while (1)
    read data for a frame
    decode data
    display frame
    evaluate resource usage
```

It is the evaluation part of this loop which will look at the resources that are being expended over time to play the video frames and decide whether the amount of work being performed should be increased, decreased, or remain the same. If it decides the work should be *locally* increased or decreased, it may change some state in the player to indicate how the data should be decoded (by looking at more or fewer bits of the data). If the evaluation phase decides that the work should be *globally* changed, it may initiate negotiations with the source of the data stream to try to increase or decrease the bandwidth of the bitstream. This negotiation may or may not change the state of the player itself, but the changes to the characteristics of the bit stream and the point in the bitstream where the change takes place should be clearly identified in the player and should be recognized in decoding the data. Thus, if and when a change in the format of the bitstream occurs, the player will be able to make the appropriate changes in decoding the stream.

So the software structure of the adaptive player, with a little more detail filled in, becomes:

```
while (1)
    read data
    check for control data
    switch based on bitstream format
        switch based on local decoding state
            decode data
    display frame
    evaluate performance
```

The application first reads the data and then checks either for control information embedded in the data or for some kind of synchronization point which is known, through information communicated via an external channel, to imply a change in data format. Then the proper algorithm is chosen to decode the data based on the format of the bitstream and on the local decoding state. Once the data is decoded, it is displayed (possibly using different algorithms indicated by the player state), and finally the performance is evaluated. To reduce overhead, the performance may not be evaluated during every iteration of the loop.

This example shows how a player might change its behavior and thus its performance characteristics based on decisions about local algorithms and global changes in data streams.

4.3.4.2 Adjusting reservation levels

Another way an application might react to noticing a difference between its usage and reservation is to change the reservation (without modifying its computation). We will examine several cases, possible behavior modifications, and their effects on delay, efficiency, and total reservation.

Under-reserved applications

As indicated in the section on measuring resource usage, an application is under-reserved if its resource usage requirements are greater than its reservation level. There are a couple of ways to change the reservation parameters to accommodate this situation:

1. increase only the units of reserved resource usage
2. increase both the units of reserved resource usage and the reservation period.

By increasing the reserved resource usage to match the computation's requirements, the application can ensure that the resources will be available to the computation, and the computation can be invoked just as often as before. The delay experienced by each computation will be decreased since there will be fewer overrun situations to cause delays, but the overall reservation level is increased. This means there is less resource capacity available for other applications, or when resource reservations are really tight, it may be impossible to increase the reservation level at all.

If the application increases the units of reserved resource and the reservation period proportionally, there will be enough reserved resource capacity in each reservation period to service the computation. And since the reserved amount and the reservation period are increased proportionally, the overall reservation level is not increased. This also implies that the computation is requested less often to correspond with the longer reservation period since requesting it just as often would not reduce the overall workload (without a load-shedding mechanism coming into play).

Over-reserved applications

Over-reserved applications are those which have a reservation level that is greater than the actual resource demand. This situation is inefficient since the application has more resource capacity reserved than it expects to use, and that reserved capacity could be used to ensure predictable performance for other applications that will actually use the resource.

The simplest action to take in this case is to reduce the units of resource reserved in each period to a value that is closer to the actual resource requirement. It is possible to increase the reservation period without increasing the period of the computations to decrease the reservation level, but that would have other undesirable effects on the timing of the program, such as increasing the delay for some computations.

4.4 Chapter summary

This chapter describes how programs should be structured to take advantage of reserves for predictable real-time performance. For hard real-time applications, information about the resources used by and timing requirements of each program must be known at design time and must be used in planning reserve allocation. So a localized way of using reserves might be appropriate. With local reserves, each program allocates its own reserves based on its requirements and the requirements of other programs that depend on it. For dynamic soft real-time systems, a global method for reserve allocation where an activity allocates the resources for all of its constituents including external servers and operating system services might be more appropriate. This makes it easier to monitor and control the usage of the entire activity rather than just localized parts of it. These recommendations are not cast in stone; the choice of whether to use localized or global reserve allocation ultimately rests with the system designer.

The discussion also dealt with methods for determining resources required by an application, reservation parameters appropriate for an application, and adaptive methods for adjusting reservation parameters or behavior based on performance history. In hard real-time systems, many of these questions must be answered at design time, and there is less flexibility in adaptation strategies. Soft real-time systems, however, have a great deal of flexibility and can take advantage of some of the techniques described in this chapter.

Chapter 5

Implementation

This chapter describes an implementation of processor reserves done using the Real-Time Mach operating system. It discusses applications that were modified to use processor reserves, network protocol processing software modified to use reserves, a QOS manager for negotiating resource allocation with applications, and tools for reserve monitoring.

5.1 Overview

This chapter describes the implementation of processor reserves in RT-Mach as well as several other components of the system. It also covers some applications that were modified or designed and implemented to use processor reserves. Figure 5-1 shows the various components and gives an indication of their relationship.

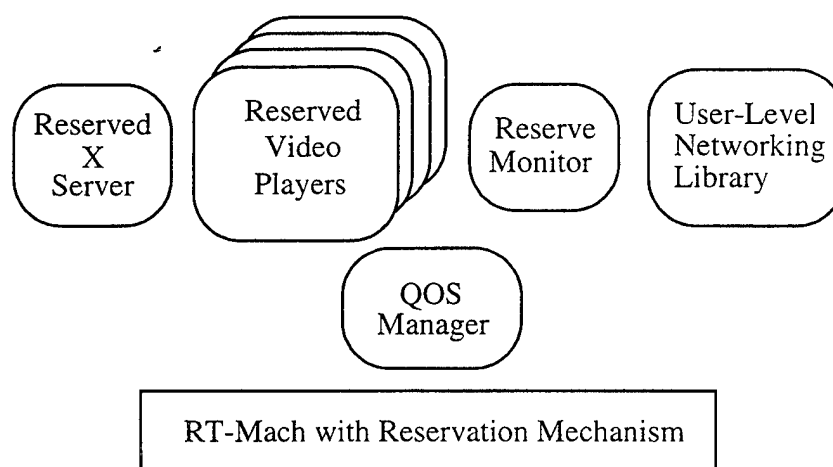


Figure 5-1: System Components

First there is the implementation of processor reserves in the RT-Mach kernel which is the basis for the rest of the implementation work. Reserves were implemented as a new kernel abstraction with operations for create/terminate, requesting reservation parameters, binding threads to reserves, and extracting usage information about reserves.

Several real applications were modified to use processor reserves including: a Quick-Time video player developed at CMU called QTPlay, an MPEG decoder called `mpeg_play` [93], and a version of the X Server [34].

A version of the user-level socket library [70] was modified to use reserves as well. This socket implementation supports predictable performance for applications that send and receive network packets.

A QOS manager was implemented to allow for more sophisticated negotiation of reserve parameters than that provided by the kernel mechanism itself. The QOS manager interacts with applications to try to balance resource usage and negotiate with applications when conflicts arise in the resource reservation requests.

Tools were implemented to help manage reserves and to monitor resource allocations and measure usage. The `rmon` application is a reserve monitor that provides a graphical user interface for reserves. It displays the reservation levels and usage in recent history, and it also allows the user to change the reservation parameters from the graphical interface.

5.2 Reserves in RT-Mach

The implementation of processor reserves in RT-Mach involved adding the new reserve abstraction, implementing the operations on reserves, creating a new scheduler, and adding code for accurate usage measurement. In addition to the new scheduler, the reserve implementation also required modifications in the RT-Mach priority inheritance mechanisms to support reserve inheritance and reserve propagation.

The reserve abstraction in RT-Mach is managed much like the other abstractions like hosts, processor sets, tasks, threads, etc. that originated in Mach 3.0 [11]. These types of resources in Mach are referenced by ports, which are used as capabilities.

In RT-Mach, processor reserves are allocated from processor sets. In the uniprocessor version, there is only one processor set, so all reserves originate from this processor set.

5.2.1 Attributes and basic operations

Abstractions like tasks and threads offer basic operations such as create, destroy, get attributes, and set attributes. The basic operations on reserves are as follows:

reserve_create(out reserve) Creates a new processor reserve and returns it as an out parameter.

reserve_terminate(reserve) Terminates the given reserve, making its reserved capacity available for other requests.

reserve_set_attribute(reserve, attr_name, attr_value, attr_value_size) Set the value of an attribute of the reserve.

reserve_get_attribute(reserve, attr_name, out attr_value, out attr_value_size) Get the value of an attribute of the reserve.

processor_set_reserves(processor_set, out reserve_list) Returns the list of reserves associated with the given processor set.

The get attribute and set attribute operations give the programmer access to some of the attributes of reserves. The externally visible attributes that reserves have appear in the following list. The data types for the attributes are given in parentheses after the attribute names; "int" is an integer, "timespec_t" specifies a time value, "mach_reserve_name_t" is a fixed length string, and "boolean_t" is a boolean flag.

name (mach_reserve_name_t) A symbolic name for the reserve.

ckpt_total (timespec_t) The cumulative total usage measured at the at last period boundary.

ckpt_time (timespec_t) The absolute time of last period boundary (when the ckpt_total value was recorded).

accum_total (timespec_t) Cumulative total usage at the current time (usually updated when accessed).

accum_time (timespec_t) The time that the total usage was last updated.

used (timespec_t) Usage charged against the reserve so far in current period.

next_period (timespec_t) The absolute time of the next period boundary (end of the current period).

period (timespec_t) The duration of the reservation period.

computation (timespec_t) The reserved computation time.

recent_checkpoint_position (int) The position in the recent_checkpoints array for the next item to be written. The array is a circular buffer.

ncheckpoints (int) The number of how many checkpoint entries currently in the array.

recent_checkpoints[MAX_CHECKPOINT_COUNT] (timespec_t) The usage values for the recent checkpoints.

recent_checkpoint_times[MAX_CHECKPOINT_COUNT] (timespec_t) The times at which corresponding checkpoint usage values were recorded.

A "checkpoint" occurs at each period boundary for each reserve. At that time, the accumulated usage is recorded along with the absolute time of the period boundary. This information can be used later by applications or monitoring tools that need information about how much usage was charged against a reserve in a particular reservation period.

There are a few other reserve attributes that are only used internally. They form part of the scheduling state for a reserve and are not available through the get attribute operation. These internal attributes are:

reserved (boolean_t) The bit that indicates whether the reserve is in “reserved mode” or “unreserved mode”.

wait_replenish (boolean_t) An internal flag indicating that the reserve has a reservation that has been depleted for the current period. The reserve is awaiting replenishment.

start (timespec_t) The absolute time at which the first period for the reserve started.

5.2.2 Reservation requests and admission control

When initially created, a reserve does not have an associated resource reservation. Getting a resource reservation for the reserve requires an additional call. The following operation allows a programmer to specify reservation parameters and request a reservation. This call is used when there is no reservation associated with the reserve, or if the programmer wishes to request a reservation with parameters that are different from the reservation associated with the reserve.

reserve_request(reserve, reservation_parameters) Requests a resource reservation to be allocated to the reserve. The caller provides the reservation parameters. Reservation parameters include desired reserved time per period, the period itself, and start time for reservation to take effect.

This operation is used to request a reservation with certain parameters. If there was previously no reservation associated with the reserve and the reservation request succeeds, then the operation returns “success” and the reservation is granted for that reserve. If the reservation request fails, the operation returns the error, and the reserve is left without a reservation.

If the reserve already had a reservation at the time the request call was made, the behavior is as follows. If the new reservation request is granted, the new reservation parameters will be associated with the reserve, and the old reservation will be freed in this process. If the new reservation request fails, the old reservation parameters remain in effect; that is, the old reservation will not be freed if the new reservation request cannot be granted.

The request operation invokes the admission control policy to determine whether the new reservation request can be accommodated given the collection of other reservations that have already been accepted for the resource. The RT-Mach implementation uses an admission test based on rate monotonic analysis, but the decision is somewhat optimistic in that it uses a utilization bound of 90% for testing for schedulability. This is based on the analysis of average schedulable bound [63], which says that for a randomly generated task set the schedulable bound is 88% on average.

5.2.3 Scheduling

The scheduler in RT-Mach was structured so as to make it easy to develop and incorporate scheduling policies. The scheduler uses a well-defined interface for scheduler operations, and a function pointer table in the scheduler contains the operations for the scheduling policy in effect. The scheduling policy can be changed dynamically by putting the ready threads in a policy-independent queue, changing the pointers in the function table to refer to the operations for the new scheduling policy, and then transferring the ready threads into a policy specific queue.

Several scheduling policies, including the "Reserves" scheduler, are supported in RT-Mach [85]. The scheduling policies in RT-Mach are associated with "processor sets," and in the case of a uniprocessor, there is only one processor set in the system. The operations to get and set attributes of a processor set are used to query or set a scheduling policy.

processor_set_get_attribute(processor_set, attr_name, out attr_value) To get the scheduling policy for a processor set, PSET_SCHED_POLICY_ATTR is specified for "attr_name". The operation sets the "attr_value" to reflect the currently active scheduling policy.

processor_set_set_attribute(processor_set, attr_name, attr_value) To set the scheduling policy for a processor set, PSET_SCHED_POLICY_ATTR is specified for "attr_name". A value such as SCHED_POLICY_RESERVES is given for the "attr_value".

Several scheduling policies are implemented in RT-Mach. The original Mach time-sharing policy is available, as are several varieties of fixed priority and rate monotonic policies. Earliest deadline scheduling is available, and round robin scheduling is supported for experimental purposes. Reserve-based scheduling is also a policy option. The policies that are available in the MK83j version of RT-Mach are:

1. Mach Time-sharing - Original time-sharing policy.
2. Fixed Priority/RR - The fixed priority/round robin policy services threads in order of a fixed priority associated with each thread. Within a priority class, threads are scheduled round robin with a quantum.
3. Fixed Priority/FIFO - The fixed priority/FIFO scheduler uses fixed priorities as well, but within a priority class, threads are scheduled using a FIFO discipline (with no quantum).
4. Rate Monotonic - Rate monotonic scheduling based on the periods given to periodic threads.
5. Deadline Monotonic - Deadline monotonic scheduling based on the deadlines given to periodic threads.
6. Earliest Deadline First - Schedules based on the deadline information of threads.
7. Round Robin - Simple round robin scheduling (with a time quantum).

8. Reserves - Scheduling policy for the reservation system.

The “Reserves” scheduling policy uses one queue for “reserved mode” threads, which are listed in order from smallest reservation period to largest. It uses an additional table of queues for implementing a multi-level feedback queue for time-sharing or “unreserved mode” threads. The reserved mode threads are scheduled first, and when there are no more reserved mode threads, the scheduler services the unreserved mode threads. In order to prevent starvation of unreserved mode threads, the reservation parameters are limited. In the implementation, a reservation cannot have a period larger than one second. This ensures that no reserved computation time can be greater than 0.9 second, so in the worst case, reserved activities can hold the processor continuously for no longer than 1.8 seconds before time-sharing programs get a chance to use the processor.

5.2.4 Usage measurement and enforcement

The scheduler for the reservation mechanism requires very accurate usage measurement so that the system can keep track of how the resource usage of each activity relates to its reservation (if any). In particular, reserved activities must be prevented from over-running their reservations and interfering with other reserved and unreserved activities.

To accumulate very accurate usage measurements, the system has code in the context switch from an old thread to a new thread that does the following:

1. takes a timestamp from a high-resolution free-running clock.
2. computes the duration of time the old thread was running and charges that usage against the reserve associated with the old thread.
3. stores the timestamp for doing the same computation later for the new thread.

Those actions mean that threads get charged for the amount of time they spent on the processor rather than getting charged an estimate of the time they spent. This timestamp method much more accurate than the method used for accumulating usage in many time-sharing systems where the process running at the time of a clock tick is charged for the duration of the clock tick (whether it was running the whole time or not).

The reserve abstraction has several operations that provide the programmer with access to usage information in the reserve. The usage-related operations are:

reserve_get_checkpoint(reserve, out_checkpoint_total, out_checkpoint_time, out_accum_total, out_accum_time) Get reserve’s checkpoint information, taken from the last reservation period boundary.

reserve_get_attribute(reserve, attr_name, out_attr_value, out_attr_value_size)
The **reserve_get_attribute** operation can be used with “attr_name” set to **RESERVE_RECENT_CHECKPOINTS** to get an array of the recent checkpoint values for the last several period boundaries.

In addition to the accurate usage measurement, the enforcement mechanism uses timers to keep threads from over-running their reserved computation time and to replenish the reserved time for a reserve appropriately. Two kinds of timers are used for doing these things: the overrun timer and replenishment timers.

The overrun timer is set at each context switch, and it is set for the maximum time the new thread could run before over-running its reserved computation time for its current reservation period. If the thread is still running when the timer expires, the system will update the reserve to show that the activity used all of its time for that period, and it will change the activity to unreserved mode. Then the scheduler will get an opportunity to reevaluate ready threads, and it may decide to switch to another thread.

If the current time is close to the end of the reservation period for the new thread and the reserved time is longer than the difference between the current time and the end of the reservation period, the overrun timer is set to expire at the end of the reservation period. If the timer goes off at that point, the reserve will be replenished and the activity will again be eligible to run.

The second kind of timer is the replenishment timer. Each reserve has a replenishment timer that is initially set at the reserve's start time. The timer is set to expire at the end of the reservation period (or the beginning of the new reservation period). When a replenishment timer expires, the system changes the state of the reserve to reflect that it has a new allocation of its reserved time for the next reservation period. The reserve is set to reserved mode, and the replenishment timer is set to expire again at the end of the new reservation period.

5.2.5 Reserve propagation

One of the key features of the reserve abstraction is that reserves can be bound to threads as appropriate for particular applications. This feature is useful in the situation where an application initiates a reserved activity that may invoke services of server processes locally or even on remote machines. When invoking a server, an application can make its reserves available for the server to use in its computations. The server can then take advantage of having the resources available, and the time it takes to perform the computation on behalf of the client can be charged to the client's reserves. In this way, reserves provide a method for consistently measuring resource usage of entire activities, even if threads in different protection domains cooperate on behalf of the broader activity.

The following operations are related to the binding of reserves to threads:

thread_set_current_reserve(thread, reserve) Each thread has a current reserve and a base reserve. The value of the current reserve may be the result of a reserve propagation, but it is not necessarily permanent. It may eventually revert to the base reserve. This primitive sets the current reserve of a thread.

thread_restore_base_reserve(thread) Makes the base reserve the current reserve for the given thread.

thread_set_reserve(thread, reserve) Set the reserve of a thread.

thread_get_reserve(thread, out reserve) Get a thread's reserve.

In addition to the binding operations, the priority inheritance mechanism of RT-Mach [59,86] aids in ensuring bounded delay for access to servers and mutexes. In the context of reserve scheduling, "priority inheritance" means "reserve inheritance" in the following sense. Interpreting "priority" in the broadest sense, one could think of a thread's reserve information and time-sharing priority information as combining to yield a total ordering for values of these fields. The scheduler schedules threads based on this total ordering from highest "priority" to lowest:

1. threads that have the "reserved" bit set are ordered with smaller reservation period having higher "priority" than larger reservation periods.
2. threads with the "reserved" bit cleared are ordered according to their time-sharing priority, which is a field of each thread (not a field of the reserve).

This concept of "priority" comes into play in the priority inheritance mechanism in RT-Mach. As an example, consider a single-threaded server with several clients. When a client makes a call to the server, the server takes on the "priority" of the client (done in the priority inheritance mechanism) and binds its own thread to the client's reserve to charge its time to it (using the bind operation).

If during this service time a second client with a higher "priority" makes a request, the IPC mechanism enqueues the new request for the server. It then calls on the priority inheritance mechanism to change the "priority" of the server to that of the newly enqueued client (thus limiting the duration of the "priority" inversion). The server continues to charge time to its first client's reserve, however, so that reserve will reflect the true resource usage required for the computation. After the service is finished, the server stops charging against the first client's reserve, picks up the request from the second client, and starts charging against the second client's reserve. The server continues to execute under the "priority" of the second client.

Priority inheritance for reserved activities presents an additional complication beyond what fixed priority inheritance mechanisms must face. In particular, with reserves (and with other dynamic priority disciplines), the "priority" the server takes may change during the service. For example, if the server executes for longer than the reserved time of its client's reserve, the reserve will be degraded into unreserved mode, and the "priority" thus changes. In the implementation, the priority inheritance mechanism is informed when this happens so that it can set the priority of the server to the appropriate value given the list of clients waiting for that server. For example, if a server uses all the reserved computation time for a particular client it would normally have its reserve downgraded and its "priority" decreased. However, if another reserved client is waiting for the server, the server will inherit the "priority" of that client so as to avoid a priority inversion.

5.3 Applications

A number of applications were modified to use reserves to show that real applications could actually achieve predictable behavior using the reservation system. A QuickTime video player and an MPEG decoder were modified to use reserves, and a version of the X Server [34] was modified to cooperate with reserved applications to provide predictable window system services.

5.3.1 QuickTime video player

A QuickTime video player, called QTPlay, was implemented at CMU. The player can display JPEG encoded video as well as raw, unencoded video. The player was modified to use reserves to achieve predictable performance.

QTPlay avoids interactions with system components that have not been modified to support predictable performance via reserves, such as the UX server. It loads a short clip of video into memory during initialization to avoid interaction with the UX server during playback. For experiments, the player loops over the clip for the duration of the test. Figure 5-2 summarizes the structure of the reserved QTPlay application.

```
load short video clip
allocate reserve with command-line parameters
create periodic threads
bind thread to reserve
...
while not done
    save start timestamp
    display a frame
    save end timestamp
...
dump timestamps to a file
```

Figure 5-2: QTPlay Outline

At initialization, QTPlay reserves time on the processor and binds the periodic thread responsible for frame processing to the reserve. The start time of the periodic thread and the start time of the reservation are synchronized so that when the thread becomes ready at the beginning of each period, the allocation of processor time will be available as well. The other reservation parameters, in particular the reserved computation time and the reservation period, are given as command-line arguments. They are typically determined by measurements made prior to the execution of the player. This particular application does not dynamically discover the appropriate reservation parameters nor does adjust the reservations after execution begins.

The player uses a version of the X Window System library, Xlib, that was modified to cooperate with a reserve-enabled X Server. This library passes a reference to the thread's

reserve when the player opens the connection to the X server. The X server then uses the reserve for operations requested by the player (such as DisplayBitmap).

In each period, the thread displays a frame of the video and then saves the start time and completion time for the frame in a buffer in memory. Just before the player exits, it dumps the contents of this timestamp buffer to a file for subsequent analysis.

5.3.2 MPEG decoder

The Berkeley MPEG decoder [93] was modified to use processor reserves in RT-Mach. This version of `mpeg_play` reserves processor capacity during its initialization and periodically evaluates its performance and makes adjustments to its processor reservation and timing constraints as necessary.

The original Berkeley MPEG decoder works by repeatedly reading MPEG encoded macroblocks from an input stream, transforming them, and displaying the frames. The underlying `mpeg` library has some features for managing the timing of frames, but the simple player that is provided to demonstrate the use of the library displays frames as fast as possible without attempting to regulate their timing.

A number of changes and extensions to the MPEG player were required to enable predictable performance and to take advantage of the timing features of RT-Mach as well as the processor reservation mechanism. Figure 5-3 summarizes the code structure of the modified version of `mpeg_play`.

```
load short video clip
allocate reserve with command-line parameters
create periodic thread
bind thread to reserve
...
while not done
    save start timestamp
    display a frame
    save end timestamp
    if frame_number mod 30 == 0 then
        evaluate usage
        adjust reservation parameters and/or algorithm
```

Figure 5-3: `mpeg_play` Outline

As with the QTPlay, `mpeg_play` prefetches a short clip of video into memory to avoid interacting with the file system during runtime. The frames are decoded and displayed by a periodic thread that has the period desired for video playback, typically 33 ms.

During the initialization of the modified MPEG player, it requests a processor reservation based on an estimate of the computation time and the length of the period. Since the computation time may vary on different hardware platforms and different MPEG data

streams, it is very difficult to get an accurate estimate before running the application, and this player tunes its reservation parameters as it executes.

For each frame, the player records the time the computation was started, the time it ended, and the amount of processor time it used during its execution (taken from the usage information in the processor reserve). It periodically computes statistics on these numbers for the recent periods to find out how much computation time was required for each frame and whether the delay for the computation is excessive. This information is used to decide what adjustments need to be made (if any). In its evaluation, `mpeg_play` distinguishes three cases:

1. reservation level okay, do nothing.
2. reservation level too low but some capacity is available to be reserved. increase reserved computation time while keeping the same period.
3. reservation level too low and no additional capacity is available to be reserved, increase the reserved computation time to the desired amount and increase the reservation period proportionally.

The modified version of `mpeg_play` incorporates some basic adaptive techniques, but it could be extended in a number of directions to improve its flexibility and performance. The decoding and display phases of the player should be decoupled to allow the variation in decoding time to be masked by buffering with the application. Incremental decoding techniques would yield several options for how much computation to do for each frame, and changing the dithering algorithm dynamically would increase flexibility as well.

5.3.3 X Server

Each real-time X client acquires a processor reserve and charges its own execution time against that reserve as well as providing the reserve to the X Server so that the Server can charge service time done on behalf of that client to the appropriate reserve. We have modified a version of the X Server to order service requests according to their timing constraints and to charge service time to the client for which the service is performed. Basically, the server should mimic as closely as possible the behavior that would be observed if each client could do its own graphical display within the context of its own address space and scheduling domain. Thus the modified X Server has the following properties:

1. The Server ensures that the activities of real-time clients are isolated from unwanted interference from non-real-time X clients by ordering all request from real-time clients down through non-real-time clients and servicing them in that order.
2. The Server itself is isolated from unwanted interference from non-real-time applications (even applications which are not X clients) by virtue of the processor reservation mechanism. The reservation system ensures that, while the X Server is running under a client's reservation, the resource capacity associated with that reservation is available to the X

Server.

3. Other real-time applications that are not X clients are isolated from unwanted interference from the X Server and its clients if they use the reservation system. This would not be true if the X Server were just assigned a "high priority" or if it over-reserved resources.

The goal of this work is to achieve predictable performance for real-time applications that make use of the graphical display services provided by the X Window System. "Predictable performance" means that real-time applications will be scheduled based on their timing requirements, and their graphical display requests serviced by the window system will be scheduled to meet the timing requirements. Thus, the abstract activity for each real-time client, consisting of the computations within each client application and the associated computations within the window system, should suffer only bounded delays due to other real-time and non-real-time applications sharing the window system.

The processor capacity reserve mechanism provides this kind of timing isolation for independent programs which do not communicate or synchronize with each other. However, when applications share a single software resource such as the X Server, the same kind of timing isolation provided by reserves must be extended into the Server's computations. To provide this isolation and bounded delay, the following is required of the server:

1. Requests from different clients that queue up in the server should be serviced in the order that the clients would be serviced if they were doing the work themselves and being scheduled by the processor reservation mechanism. In other words, the server should handle requests in order of client "priority" (where client priority refers to an implied ordering among clients defined by the reservation system).
2. Computation performed in the server on behalf of a client should enjoy resources, such as processor capacity, that have been reserved for that client. The server should execute at the priority of the client whose request it is servicing. Likewise, the resource usage for such a computation should be charged to the client's reservation, so that a client is prevented from getting more than its reserved time by sending some work to the server and then doing other work locally.
3. Priority inversion should be minimized (and unbounded priority inversion completely avoided) in servicing the clients' requests. Thus if the X Server is occupied with a request from a client when another request comes in from a higher priority client, the server should inherit the priority of the newly arrived client.

These requirements have many implications for the coding of the server. The idea that the server should mimic the behavior of individual threads performing the same computations places some restrictions on how the server can be designed. Also, each of the three specific requirements listed above has some additional implications for the coding of the server.

First we address the desire to have the server behave as the individuals would. This is conceptually easy to achieve by thinking of spawning a new thread for each client's request as it arrives at the server, binding the thread with the reservation or priority of its client, and then allowing the threads to be scheduled by the processor reservation system based on that information. Unfortunately, spawning a potentially large number of new threads is expensive, and while there exists a version of the X Server that is multi-threaded [110], the one on which this work is based has only a single thread. With a single-threaded server, we try to mimic the desired behavior by satisfying the three requirements listed above as follows:

1. Client requests are enqueued in the server in priority order.
2. At the beginning of the computation for each service request, the server takes on the resource allocation persona of the client, enjoying the resource reservations of the client and charging usage against the client's reserve.
3. The RT-IPC [59] mechanism handles priority inheritance to minimize the effects of priority inversion.

These are the modifications made to the X Server to provide predictable performance. However, there are some problems with the X Server that interfere with real-time applications and which are very difficult if not impossible to fix. Several of these problems are addressed in the development of a window system intended for real-time performance [105]. Briefly, the problems are:

1. The X Protocol supports a "grab server" operation which blocks out all other operations for an unbounded period of time.
2. The X library batches requests for higher throughput. This can increase the delay of single operations as multiple operations are combined into one.

Despite these hindrances to 100% guaranteed real-time performance, the modified X server can provide good real-time behavior for typical multimedia applications such as video players.

5.4 Reserved network protocol processing

A predictable network service depends on how the protocol processing for network packets is handled as well as how these activities are scheduled. This section examines several different approaches to protocol processing software design and discusses the advantages and disadvantages of these approaches.

5.4.1 Software interrupt vs. preemptive threads

Traditionally, protocol processing software has been designed to take packets from the network interface and immediately begin processing them at high priority. For example, 4.3

BSD protocol processing is done at a “software interrupt level” which executes at a higher priority than any schedulable activities in the system (like processes) but at a lower priority than hardware interrupts [62]. Unfortunately, network packets associated with a low priority activity may flood the protocol processing software and execute while higher priority processes are delayed. This is an example of priority inversion [48,75].

To prevent this kind of priority inversion, it is necessary to associate priorities with packets so that they can be queued and serviced in priority order. It may also be helpful to be able to preempt the processing of one low priority packet in favor of a higher priority packet, especially if the computation time required for protocol processing is significantly more than that required for a context switch. One approach, used in the ARTS real-time kernel, has preemptible threads to shepherd packets through the protocol software [124]. This is similar to the method used in the *x*-kernel [45], but unlike the *x*-kernel threads, ARTS protocol processing threads were preemptive. This approach provides fast response to high priority packets and prevents low priority network activities from interfering with high priority work on the processor.

5.4.2 Mach 3.0 networking

Networking in the context of the Mach 3.0 UX server [36] is accomplished by calling the 4.3 BSD networking primitives, which are handled by the UX server. The UX server interacts directly with the network device drivers to send and receive packets.

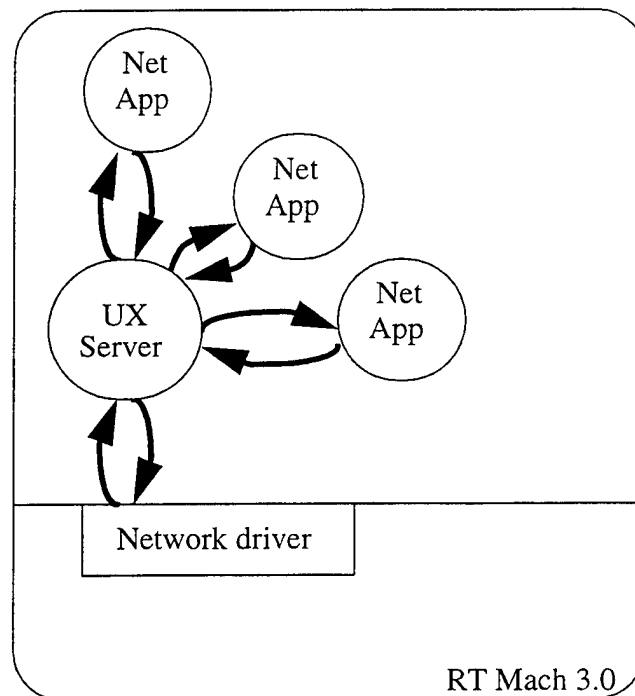


Figure 5-4: Networking with the UX Server

As shown in Figure 5-4, this makes the UX server a single point of contention for all activities that are using the network. Unfortunately, the networking code inside the UX server does not support priority. So this software does not satisfy the requirements for priority and preemptibility in predictable protocol processing software.

Another problem with networking under the UX server of Mach 3.0 is that the interprocess communication (IPC) required between the application and the UX server and between the UX server and the network device drivers adds overhead to network communication. This decreases throughput and increases latency. To alleviate these problems, Maeda and Bershad created a library implementation of TCP/IP and UDP/IP sockets [70]. Their library handles the protocol processing for sending and receiving packets and interacts with the network packet filter [139] and network device drivers directly. The library can be linked in with applications that use the networking calls, so each application can do its own protocol processing in its own scheduling domain (i.e. within its own threads). The library only interacts with the UX server to create and destroy connections and for a few other control operations. The fast path for sending and receiving packets is confined to the library itself (and the device drivers). Figure 5-5 illustrates their networking software structure.

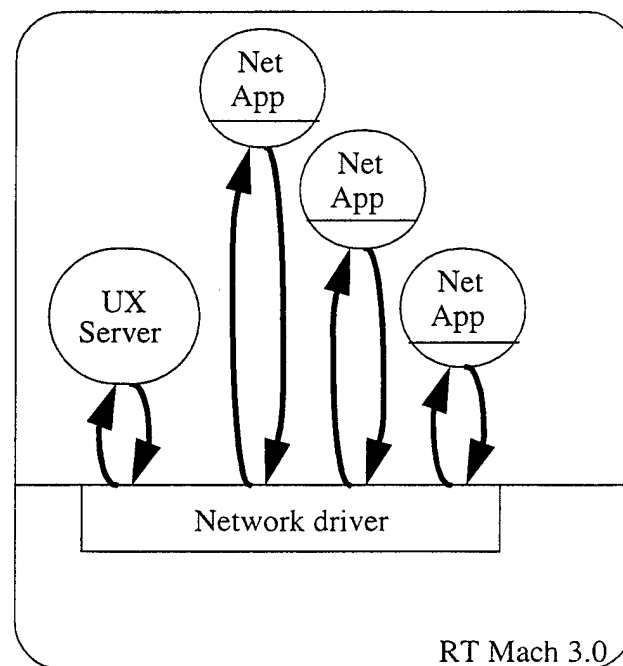


Figure 5-5: Networking with the Socket Library

Maeda and Bershad report that their socket library yields much better performance in terms of throughput and delay than the UX server sockets implementation [70]. Coincidentally, their implementation also satisfies the requirements for effective real-time scheduling

of protocol processing. By including the code in a user library, the computation is done by the user thread at the user's priority. It is also preemptible since it runs in user mode and shares nothing with other threads in other applications.

5.4.3 Reserved protocol processing

Since the socket library enables the protocol processing computation to be scheduled under the priority of the application and since it is also preemptible, the processor reservation system can be applied to programs which do socket-based communication [77]. Compared with a UX server socket implementation, the library partitions the data structures and control paths of all of the networking activities and places them in independent address spaces where they do not to interfere with each other. In the UX server, these different activities are forced to share the same queues without the benefit of a priority ordering scheme. Other activities such as file I/O, asynchronous signals, etc. may interfere with the protocol processing, thus delaying packets as a result of other operating system activity that is not even related to networking.

In the socket library, these components do not interfere with each other, so the reservation mechanism is free to make decisions about which applications should receive computation time and when. The control exercised by the reservation scheduler is not impeded by additional constraints brought on by the sharing of data structures and threads of control. Applications that use the socket library with the reservation mechanism should therefore achieve very predictable networking behavior.

5.5 QOS manager

A QOS manager was implemented to provide a central point for resource allocation decisions. It exports an interface that allows the application programmer to create and terminate reserves, to request a reservation at a specific desired level, and to set preferences for the minimum reservation level. If the reserved load becomes high and the server has difficulty granting minimum reservation levels for new requests, the server begins to downgrade some of the previously granted reservations to their minimum levels in order to admit the new reservation request at its minimum level.

In general, information about resource allocation requirements may come from a variety of sources and may change over time. Resource allocation information can come from applications themselves which may request resource and negotiate if the request cannot be satisfied immediately. It can come from static user preferences about which applications should be more resources under what circumstances. And it can come from various user interface elements designed to bring resource management decisions to the console user.

5.5.1 Information sources

The QOS manager uses the information it gathers to make policy decisions about how to allocate resources to various activities. The information may come from user preferences

files, applications themselves, and graphical resource management tools. Figure 5-6 summarizes the information flow associated with the QOS manager.

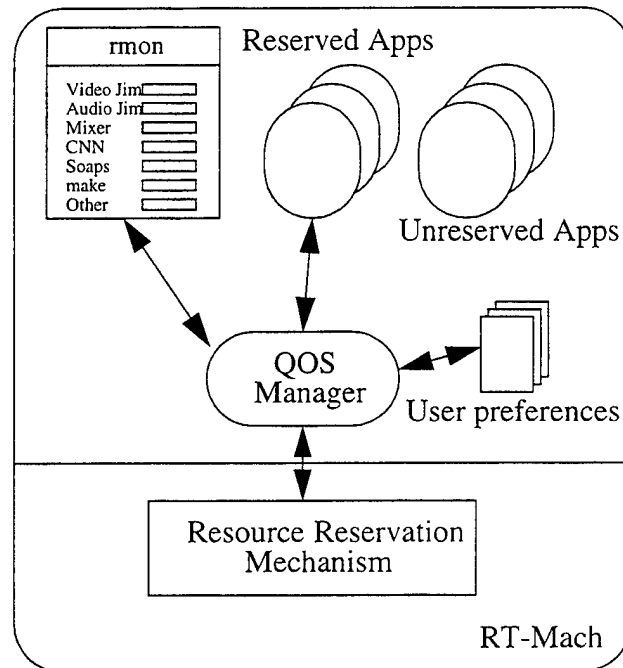


Figure 5-6: Resource Management Schematic

The static user preferences used by a QOS manager might come from a configuration file located in the user's home directory or in a system default directory. Such a file could contain arbitrarily sophisticated rules for the QOS manager to use in making allocation policy decisions. For example, the file might contain rules to indicate how the user's focus should affect resource allocation. It might have rules to determine which applications are more important (e.g. specifying that audio/video applications are more important than file transfer). There might be rules about how temporal properties indicate which applications are more important (e.g. giving recently created applications preference over older applications). And there might be rules about how past usage should affect future reservation.

The dynamic user preferences might come from the applications themselves, from a separate tool, or from some mechanism associated with a window manager. In any case, cues given by the user, which can be picked up in the user interface, are very important to the policy decisions that must be made about where to allocate resource capacity. These cues can be explicit, where the user makes certain gestures to change the resource capacity allocations of various activities. Or the cues can be implicit, as in the case of a window manager

which notices which window has the user's focus (based on the position of the mouse pointer) and passes this information along to the QOS manager.

Information about the recent resource usage of various activities might be used to determine what the future resource reservation levels should be for those activities. For example, an audio player receiving transmissions over the network might become quiet due to long-lasting lull at the sender. When this happens, it may be appropriate to notice the lack of resource usage in the associated reserve and temporarily scale down the reservation level in order to free up more reservable capacity for other activities. A QOS manager with this feature would undoubtedly also provide a mechanism for such dormant applications to come back to life at their original reservation level once they become active again.

5.5.2 Admission control

The admission control policy of the QOS manager must be coordinated with the admission control of the system. The reservation system has an admission control policy that allows it to enforce reservations and keep itself internally consistent with respect to resource allocation and enforcement. The QOS manager must have a version of the same admission control policy so that it can evaluation reservation requests that it gets and look at more sophisticated issues such as how different requests it gets can be combined or changed to fit together better.

This design was chosen because it keeps the admission control test of the kernel simple and fast while allowing arbitrarily sophisticated admission control decisions and negotiations to be carried out in user-level QOS managers. It would be possible to combine the two policies but there are drawbacks to that approach. If the sophisticated policy with negotiation were implemented in the kernel, the system would become more complicated, slower, and less flexible. If the kernel depended on user-level admission control for its own consistency, it would be vulnerable to errors in the user-level QOS managers.

5.5.3 Extensions

The QOS manager reacts to new reservation requests that strain the available resource capacity by trying to free up resource capacity from among previously reserved activities, subject to the limitations that those activities allow as expressed by their minimum reservation levels. This policy could be extended to accommodate information about which activities should be downgraded first, whether new minimums could be negotiated with activities to free up even more capacity, or whether the activities requesting reservations should be denied to keep the previously reserved activities at their current reservation levels [79]. Another extension might upgrade reservations to the old desired values once reservable resource capacity became plentiful.

5.6 Tools

Two tools were developed in the course of this dissertation work to do debugging and execution monitoring for experiments and to provide a user interface for reserves. One is a reserve monitor with a graphical user interface, and the other is a usage monitor that operates in batch mode to gather usage statistics for experiments.

5.6.1 Reserve monitor

A reserve monitor, called *rmon*, provides the user at the console with a graphical user interface to monitor and control processor reserves. The two important aspects of this tool are its presentation of usage information and its support for control of resource reservation.

5.6.1.1 Usage information

The primary view of *rmon* displays basic information about all of the processor reserves in the system. This information consists of the name of the reserve, a graphical representation of the recent usage information, normalized to the reservation period, and the reservation period itself. Figure 5-7 shows a screen dump of this primary view.

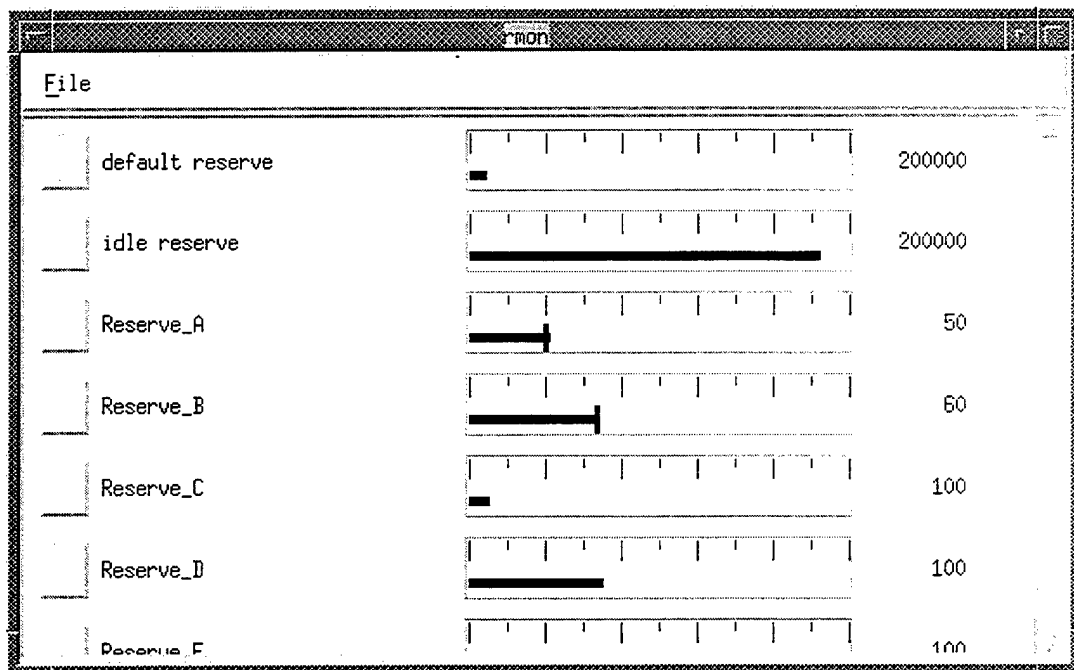


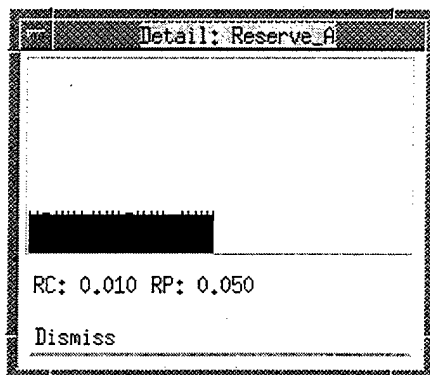
Figure 5-7: *rmon* Main View

As the figure shows, each reserve takes one row in the display. Each row contains the following elements:

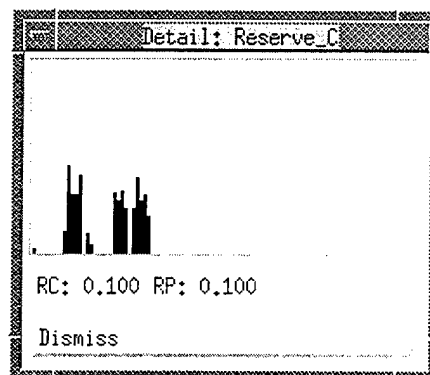
- detail button - Pressing the detail button pops up another window which shows more detailed information associated with the reserve.
- reserve name - Each reserve may have a name associated with it for ease of identification.
- graphical usage display - This graphic displays a bar with length corresponding to the percent resource usage over the last several reservation periods. The usage is normalized to the reservation period, and the graphic includes markings to indicate the scale of the usage.
- reservation period - The reservation period indicates the averaging interval of the usage measurement.

As reserves are created and terminated in the system, corresponding rows are created and destroyed in the primary view. The two system reserves (called “default reserve” and “idle reserve”) always exist. So they always appear in the view. The default reserve is where all the usage is charged for applications that do not have their own private reserves. It has no actual resource capacity reservation associated with it; it just accumulates the usage of the unreserved programs. The idle reserve accumulates the usage of the idle thread; it also lacks an actual resource reservation.

For each reserve that has a resource capacity reservation associated with it, rmon displays a vertical bar in the usage graphic to indicate the level of the reservation, in terms of normalized capacity. In Figure 5-7, the reserves named “Reserve_A” and “Reserve_B” have resource capacity reservations associated with them whereas “Reserve_C” and “Reserve_D” do not. The vertical bars in the usage graphics of Reserve_A and Reserve_B indicate that they have reservations of 20% and 33%, respectively. The reservation periods are 50 ms and 60 ms, respectively.



(a)



(b)

Figure 5-8: rmon Detail Views

Pressing the “detail” button at the beginning of a reserve’s row pops up another window of detailed information about the reserve. This includes a graphical display of the recent history of resource usage as well as the parameters of the reserve. Figure 5-8 shows two example detail windows. Part (a) of the figure shows the detail for an activity that has a reservation, and part (b) shows an activity without a reservation.

As shown in the figure, the recent history occupies the top portion of the detail window. It shows the normalized usage of the reserve over the last several reservation periods, and it advances in real-time in a manner similar to that of xload [74]. For the reserved activity in Figure 5-8(a), which is Reserve_A from the Main View, this usage is fairly constant over time. For the unreserved activity in Figure 5-8(b), Reserve_C, this usage is variable from period to period. Each window also displays the reserved computation time and the reservation period. The unreserved activity has zero reserved computation time but a non-zero reservation period (the implementation represents an unreserved activity using a reserved computation value equal to the reservation period, and this representation happens to be exposed in this view). This indicates that usage measurements for this activity will be taken based on the reservation period, but that there is no actual resource capacity reservation.

5.6.1.2 Allocation Control

The level of the reservation for a reserve, as indicated by the vertical bar in the normalized usage graphic, can be changed by clicking the mouse in that usage graphic at the level desired for the reservation. This action modifies the reserved computation time parameter of the reserve without changing the reservation period.

The upper screen view in Figure 5-9 shows several reserves at various reservation levels. Notice the position of the mouse pointer in the usage graphic of the reserve called Reserve_B, which is reserved at 20% of processor capacity. Clicking the mouse button with the pointer at this position changes the reservation level of Reserve_B to that shown in the lower screen dump in the figure. The reservation level is now about 40%, and the actual usage of the activity reflects the availability of that additional capacity.

5.6.2 Usage monitor

A usage monitor based on reserves was developed to aid in debugging and to support usage measurements for experiments. This monitor allocates a reserve and requests a reservation for its own execution. It periodically polls for the usage on specified reserves in the system, saving the usage numbers in a large buffer. Then it formats the usage information and writes it to a file for processing by a graphing tool.

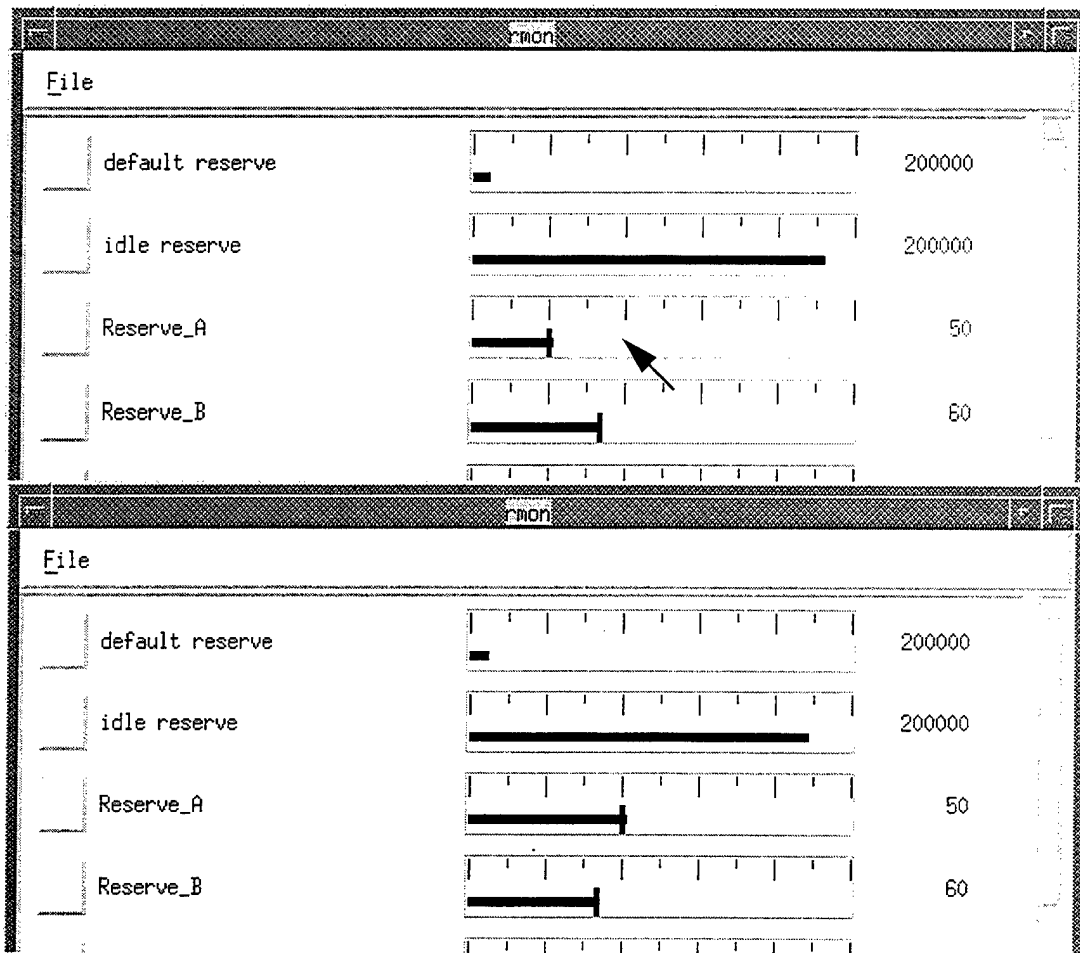


Figure 5-9: Modifying a Reservation

5.7 Chapter summary

This chapter describes the implementation several software components in the reservation system including:

- processor reserves in RT-Mach,
- reserved video players and a version of the X Server that uses reserves,
- a version of the Mach 3.0 socket library implementation modified to use reserves,
- a QOS manager,
- a tool for providing a user interface to the reservation system.

The description of the implementation of reserves presents operations that are supported for manipulating reserves as well as a description of how the scheduling and usage enforcement is handled. The implementation of two video players that use reservation is described along with that of a version of the X Server that was modified to use reserves. The section on the socket library implementation discusses issues in the organization of protocol processing software to support real-time packet processing. A description of the QOS manager indicates how it works and its relationship with applications and the reservation mechanism. Finally, a graphical user interface tool is described; it displays information about reserves and provides an interface for controlling reservation parameters.

Chapter 6

Experimental Evaluation

This chapter presents an experimental evaluation of the implementation of processor reserves in RT-Mach. The reservation mechanism was designed to support predictable behavior for real-time and multimedia applications, so the evaluation answers the questions: Can reserved programs achieve predictable behavior, and what is the price of predictability?

6.1 Overview

The experimental evaluation presented in this chapter answers two questions: Can reserved programs achieve predictable behavior, and what is the cost for predictability? These questions are addressed using synthetic benchmarks, real applications, and measurements of individual mechanisms. The chapter is divided into two main sections, one to address predictability and another to address scheduling costs.

The section on predictability shows that for a wide variety of task sets, real-time tasks exhibit predictable behavior and meet their timing constraints:

- Independent synthetic workload measurements show that for pure computations that have no interactions with other tasks, the reservation mechanism successfully guarantees timing constraints.
- Client/server experiments show that the reserve propagation mechanism helps guarantee the client's timing constraints, even when there are multiple clients with and without reservations.
- Results of experiments with the QTPlay QuickTime video player and the X Server show how this client/server pair is coordinated to meet the timing constraints of the video player, even when there are unreserved X clients competing for the attention of the X Server.
- Experiences with the mpeg_play decoder and the X Server demonstrate how an application can start with an inaccurate estimate of required com-

putation time and then adjust its reservation parameters to balance its usage requirements with the resource availability.

- Experience with a library-based network protocol software structure shows that protocol processing for real-time applications can be guaranteed using processor reserves.

The other main section of this chapter explores the scheduling costs of the reservation system. Two measurement techniques are used:

- A comparison of the system scheduling costs for periodic real-time programs that use reserves vs. periodic programs that do not use reserves shows that the cost varies, as expected, depending on the period of the program.
- Measurements of various internal operations such as reserve switch time, overrun timer handling, replenishment timer handling, and usage checkpoint operations provide a means of estimating scheduling cost for reserved task sets.

The measurements for most of these experiments were taken using RT-Mach version MK83j with UNIX server UX41. The mpeg_play and libsockets experiments used RT-Mach version MK83i, which does not differ significantly from MK83j in the features used in the experiments. The hardware platform for the first three sets of experiments was a 90MHz Pentium with 16 MB RAM and an Alpha Logic STAT! timer card. The timer card has a 48-bit free-running clock with 1 μ s resolution, and a 16-bit interrupting timer with 1 μ s resolution. For the remaining experiments, the hardware platform was the same except the processor was a 486 DX2 instead of a Pentium. For easy reference, the chart below summarizes which platforms were used for which experiments.

Experiment	RT-Mach Version	UX Version	Processor
Independent task sets	MK83j	UX41	Pentium
Client/server task set	MK83j	UX41	Pentium
QTPlay/X Server	MK83j	UX41	Pentium
mpeg_play/X Server	MK83i	UX41	486
libsockets	MK83i	UX42	486
Aggregate scheduling costs	MK83j	UX41	486
Micro measurements	MK83j	UX41	486

Table 6-1: Summary of Testbed Platforms

In general, the switch from the 486 to the Pentium speeds up the compute-intensive applications by about 30%. Since the micro measurements involve kernel instruction streams that access external devices such as the clock/timer card, these measurements are not expected to change significantly using a Pentium processor.

In summarizing the results of many of the experiments, percentiles are used to specify dispersion. While running these experiments on a desktop computer connected to the normal departmental network, occasional anomalies occurred. 5-percentiles and 95-percentiles are used to indicate the range of the strong majority of measurements while ignoring the occasional anomaly. As defined in Jain's book [50], the 5-percentile is obtained by sorting a set of n observations and taking the $[1 + n(.05)]$ th element in the sorted list (where $[.]$ is used to indicate rounding to the nearest integer). The 95-percentile is the $[1 + n(.95)]$ th element in the sorted list.

6.2 Predictability

What is meant by "predictability?" In the context of this work, a predictable application is one whose timing behavior can be determined from the application code, the resource reservations that it acquires, and its dependence on other programs. In particular, a predictable application that is not under-reserved will meet all of its timing constraints.

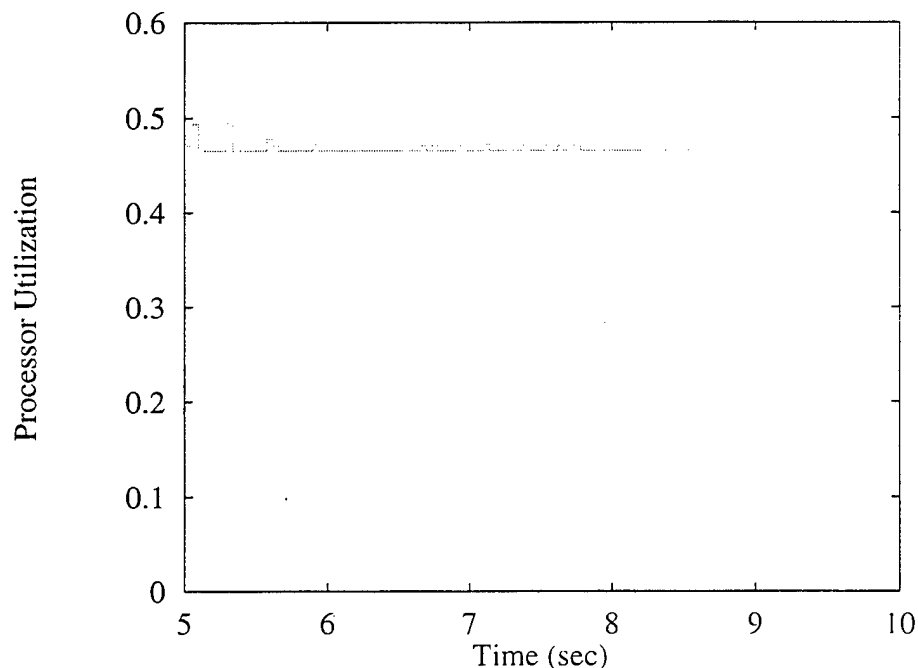


Figure 6-1: Compute-Bound Periodic Task with No Competition

As an example, consider a periodic application that computes for a fixed duration of time in each period and spends all of its computation time in a tight loop. Such an application should be able to allocate a reservation for its computation time, and it should exhibit the same behavior when it is executing concurrently with other activities as when it is executing in isolation. That is, it should be able to consume its reserved computation time in each period before the “deadline” at the end of the period.

Figure 6-1 shows the processor usage over time of a periodic application with a local computation and no competition for resources. The x-axis is time measured in seconds; it shows several seconds of usage information for the application. The y-axis is the normalized processor utilization of the application. Time on the x-axis is divided into intervals that correspond to the period of the application, and the processor time used during each period is measured and the utilization computed for the period. The utilization is plotted at that constant level for each period. For the duration of the test shown, the periodic application has an average utilization of 0.467. The distribution of measurements is very closely packed around this average with a 5-percentile of 0.465 and a 95-percentile of 0.473.

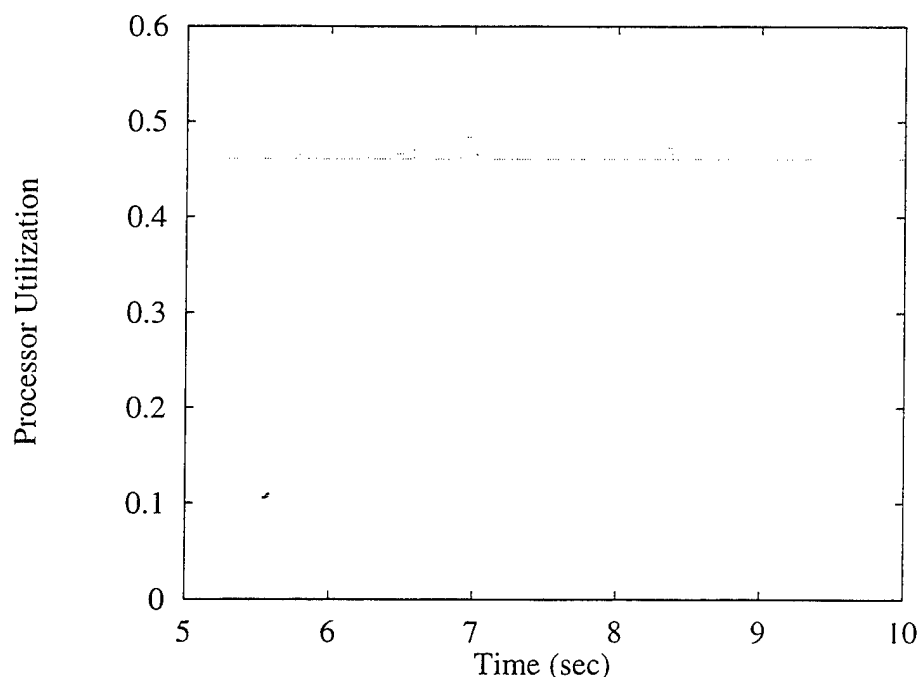


Figure 6-2: Compute-Bound Periodic Task with Competition

Figure 6-2 shows a similar graph of the processor usage of a periodic application that has a local computation but has competition for the processor from other programs (not shown). Even though there is competition, the reservation system ensures that the appropriate amount of processor time will be available to the application in each period. The application consumed an average of 0.462 of the processor in each period. As in the previous case,

most of the measurements were very close to the average; the 5-percentile is 0.460 and the 95-percentile is 0.470.

6.2.1 Independent synthetic workloads

Independent synthetic workloads are used to test whether the reservation system can successfully provide access to reserved processor resources. The example above demonstrates that the reservation system can ensure predictable behavior for a periodic application running with competition from other activities, and one of the experiments described below shows that multiple independent reserved applications can achieve predictable behavior, even with competing unreserved activities. Two additional experiments show that the reservation guarantee is independent of the number of competing activities, regardless of whether the competitors are reserved or unreserved.

6.2.1.1 Methodology

These experiments were run using two software tools developed for performance evaluation. A configuration manager parses the specification of a task set with timing parameters and reservation parameters and then creates programs with the appropriate parameters. Several different kinds of programs that exhibit different kinds of behavior can be specified in the task set. Each of these programs takes a start time, a duration to compute, a thread period, a computation time to reserve, and a reservation period. Two programs are used in these experiments:

- `arith` - Creates a periodic thread that executes in a tight loop for some duration of time in each period.
- `monitor` - Records the usage charged to reserves in the experiment. This program has a reservation of its own to enable it to run even when there are many reserved programs in the experiment.

A usage monitor is usually included in the task set to take usage measurements for all of the programs created by the configuration manager. The monitor buffers the measurements during the course of the experiment and then formats the data and writes them to disk after the experiment is completed. The data are then graphed.

Experiment 1 is designed to show that reserved activities are able to execute their periodic computations within their time constraints, even with competing unreserved activities. Even if the reservation parameters have different computation times and different periods the timing constraints of the reserved activities will be satisfied.

Table 6-2 shows the programs used in Experiment 1 along with the number of instances of each program, the timing parameters, and the reservation parameters. In this experiment there were 3 `arith` programs that were reserved with different timing and reservation parameters. One had a reservation of 5 ms of every 20 ms, the second had a reservation of 14 ms every 40 ms, and the third a reservation of 8 ms every 50 ms. The reservation is set slightly higher (1 or 2 ms) than the computation time that would be consumed by the program in isolation. This accommodates variation in the computation time due to cache effects

and context switches. In addition to those three, there were 5 arith programs running in infinite loops with no reservations; these provide compute-bound competition for the reserved activities. And finally, the experiment included a monitor program to collect usage numbers throughout the duration of the test. This monitor had 2 ms reserved of every 20 ms.

Program	#	Program Computation	Program Period	Reserved Computation	Reservation Period
arith	1	4 ms	20 ms	5 ms	20 ms
arith	1	12 ms	40 ms	14 ms	40 ms
arith	1	6 ms	50 ms	8 ms	50 ms
arith	5	infinite loop	N/A	0 ms	40 ms
monitor	1	N/A	20 ms	2 ms	20 ms

Table 6-2: Experiment 1 Parameters

The other two experiments measure the sensitivity of reserved applications to competition. Both experiments consist of eight series of tests. Each test has a reserved arith program whose measurements are the focus of the test. The series differ in that the reserved arith program increases in reserved utilization in each series. Each series itself consists of a sequence of tests with an increasing number of competitors. In each series of Experiment 2, the one reserved arith program competes with an increasing number of unreserved arith programs. For each test, the 5-percentile and 95-percentile for the resource usage measured in each reservation period is reported. The parameters for an example task set in Series 1 of this experiment appear in Table 6-3.

Program	#	Program Computation	Program Period	Reserved Computation	Reservation Period
arith	1	3 ms	40 ms	4 ms	40 ms
arith	2	infinite loop	N/A	0 ms	40 ms
monitor	1	N/A	20 ms	2 ms	20 ms

Table 6-3: Example Parameters for Experiment 2

The task set that appears in Table 6-3 has one reserved arith program that computes 3 ms in every 40 ms and has a reservation of 4 ms for every 40 ms. It also has two unreserved arith programs and a monitor program. Other tests in the Series 1 have unreserved

competitors ranging in number from 0 to 9 competitors. This task set is designed to show that regardless of what the reserved utilization is and regardless of how many competitors there are (ranging from 0 to 9 compute-bound, unreserved competitors), a reserved activity will always be able to get its reserved allocation.

Experiment 3 is like Experiment 2 except that the competitors are reserved instead of unreserved. For convenience, the competitors all have identical reservations, so the number of competitors for a reserved activity is limited to the number of competitors that can be accepted by the admission control policy.

The tests of Experiment 3 are organized into 8 series with 10 tests, just as in Experiment 2. Again, the series differ in that the reserved `arith` program that is observed varies in its reserved utilization, and within each series, the number of competitors ranges from 0 to the highest number that can pass admission control along with the observed program. Variation in the usage of the measured reserved program is again characterized by the 5-percentile and 95-percentile. The parameters for an example test in this experiment appear below.

Program	#	Program Computation	Program Period	Reserved Computation	Reservation Period
<code>arith</code>	1	3 ms	40 ms	4 ms	40 ms
<code>arith</code>	2	2 ms	30 ms	3 ms	30 ms
<code>monitor</code>	1	N/A	20 ms	2 ms	20 ms

Table 6-4: Example Parameters for Experiment 3

The task set in Table 6-4 has one `arith` program with computation duration 3 ms and a period of 40 ms. The reservation given to this program is 4 ms every 40 ms. The table lists two other reserved `arith` programs with 2 ms computation time and 30 ms period, and these both have reservations of 3 ms every 30 ms. This is a test from Series 1 of Experiment 3, and other tests in this series have the different numbers of competing reserved programs. The number of competitors for Series 1 ranges from zero to seven, but the number of competitors for Series 8 is zero since no competitors could be admitted once the primary reserved program and the reserved `monitor` pass admission control. The purpose of this task set is to demonstrate that regardless of what the reserved utilization of the primary reserved program and regardless of the number of competitors, the primary reserved activity will get its reserved allocation virtually all of the time.

6.2.1.2 Results

The results from Experiment 1 demonstrate that multiple reserved programs meet their timing constraints, despite the competition between the reserved activities and competition from unreserved activities. Figure 6-3 shows a graph of the behavior of the three reserved programs in Experiment 1, leaving out the usage measurements of the competing unre-

served activities. These usage measurements are in the same format as the example case described earlier: the x-axis is time in seconds for the test, and the y-axis is processor utilization. The usage for each reservation period for each reserved program is computed and plotted on the graph, yielding three functions of utilization over the duration of the test.

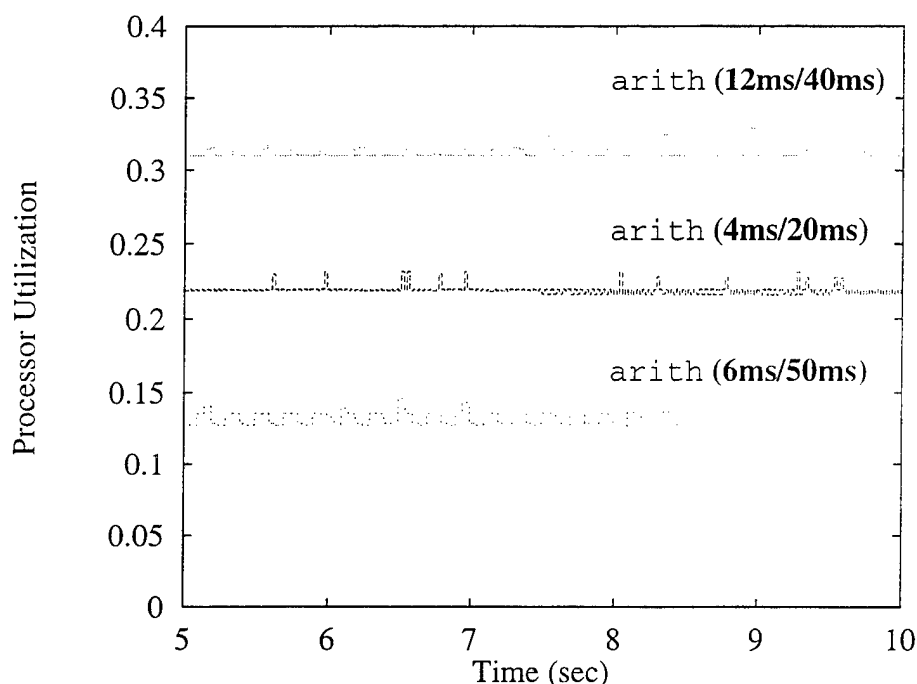


Figure 6-3: Experiment 1 Results

Each of the three reserved programs sustains a fairly constant utilization level throughout the entire test, in spite of the competition from reserved and unreserved activities. The reserved program with a computation time of 4 ms every 20 ms gets a fairly constant utilization with an average of 0.219 (context switching overheads and cache effects push the measured usage higher than what it would be in a quiescent system). The 5-percentile is 0.216 and the 95-percentile is 0.227, indicating that very few measurements fall far from the average. The reserved program computing 12 ms every 40 ms gets an average utilization of 0.312. It gets a 5-percentile of 0.310 and a 95-percentile of 0.320, so there is clearly very little variation in the utilization across periods. The program computing 6 ms every 50 ms gets an average utilization of 0.132 across the periods shown above. The 5-percentile is 0.126 and the 95-percentile is 0.141. Thus Experiment 1 shows that the reservation system can guarantee the timing constraints for multiple reserved programs even when there is competition from unreserved activities.

The results from Experiment 2, illustrated in Figure 6-4, show that the timing behavior of a reserved program is not affected by the number of unreserved competitors, regardless of the utilization of the reserved program. The graph in Figure 6-4 has the number of competitors on the x-axis and processor utilization on the y-axis. The data from the eight series are

plotted as functions on the graph. For example, the function for Series 1 starts with the average utilization for the test that has zero competitors. The function then continues to the average utilization for the test in that series that has 1 competitor and so on up to the average utilization for the test with nine competitors. At each point where the average utilization is plotted, there is also a range that gives the 5-percentile and 95-percentile to indicate the variation in the behavior of the reserved program on that test. The rest of the functions are similar. This graph shows that for each series, the average processor utilization is nearly constant, regardless of the number of competitors. Furthermore, the variation given for each measurement is quite small, indicating that for the vast majority of reservation periods, the reserved program is able to meet its timing constraints.

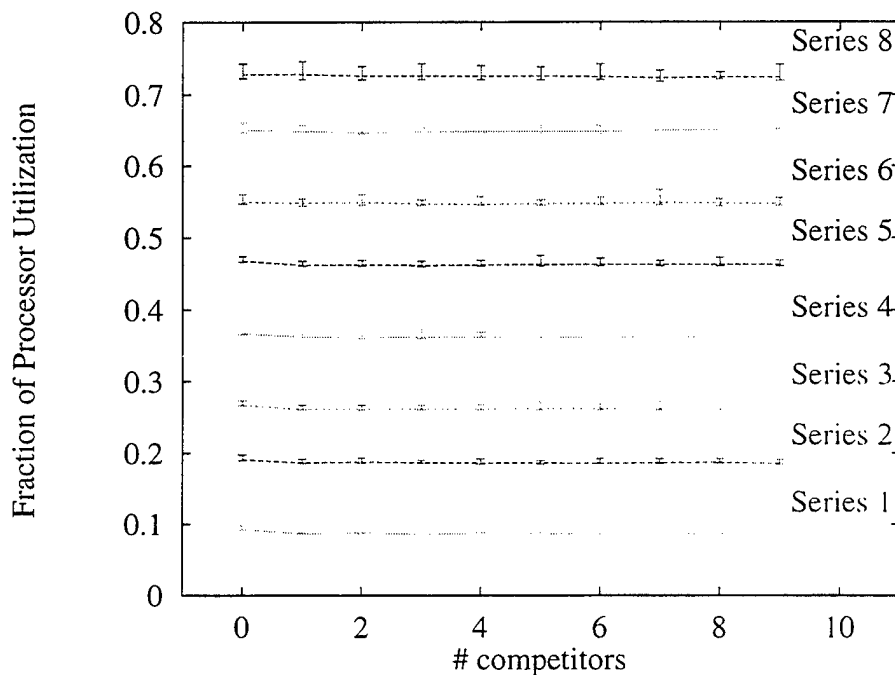


Figure 6-4: Experiment 2 Results

The results for Experiment 3 are presented in Figure 6-5. These results show that the behavior of a reserved program does not depend on the number of competing reserved activities, regardless of the utilization of the reserved program of interest. The graph for Experiment 3 is much like the graph for Experiment 2. The x-axis is the number of competing programs, and the y-axis is processor utilization. There are data from the same kinds of series, and the plot of average utilization as a function of number of competitors is the same. Each plotted point has a 5-percentile and 95-percentile range to indicate the variation. Since each of the competitors must pass the admission control policy, the number of competitors becomes more limited as the utilization of the measured reserved activity gets larger. So the maximum number of competitors for Series 1 is seven and for Series 8, no competitors can

pass admission control after the measured reserved activity and the monitor do. These results show that the average processor utilization for each series is nearly constant, i.e. it does not depend on number of competitors. The variation in processor utilization is very small, indicating that the reservations are available to allow the reserved activity to satisfy its timing constraints. This is true regardless of the processor utilization of the measured reserved program.

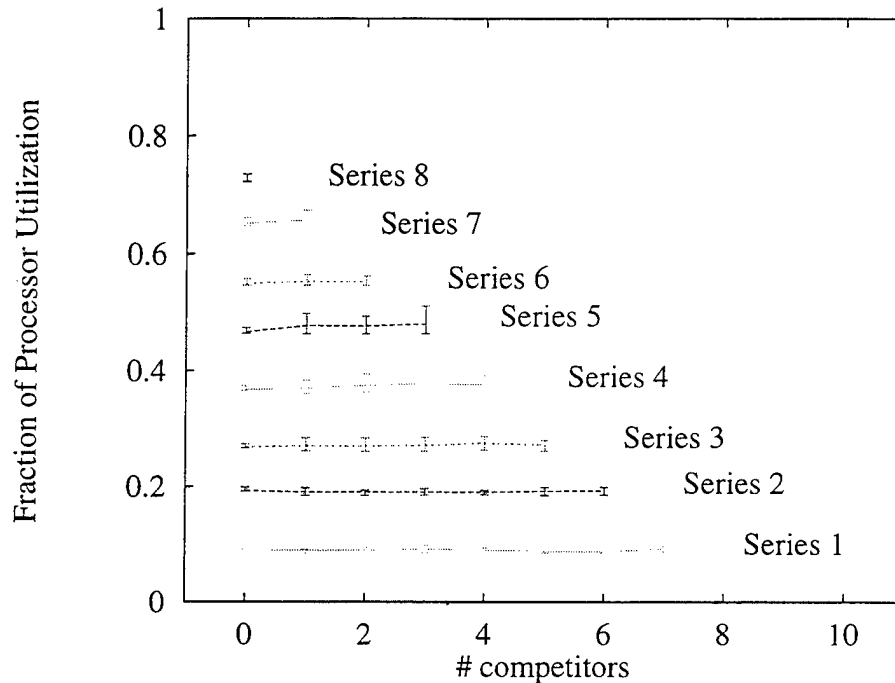


Figure 6-5: Experiment 3 Results

6.2.1.3 Analysis

These three experiments show that for cases where periodic threads allocate reserves for their computations, the reservation system is able to ensure that the reserved time is available as promised. The reserved time is available even when there are multiple reserved activities and unreserved competitors as in Experiment 1. Experiment 2 showed that a reserved activity is assured of being able to use its reserved time regardless of the number of unreserved competitors and regardless of whether the reservation is for a small computation time or a large computation time. Experiment 3 demonstrated that a reserved activity will get its reserved time regardless of the number of reserved competitors it has, and this is also true whether the reserved activity has a small amount of time reserved or a large amount of time.

Two issues are highlighted by these experiments. One is that the computation time for the reserved programs was always less than the reserved computation time by 1 to 2 milliseconds. The computations that the programs will execute are based on arithmetic computa-

tions that were timed on a quiescent system with only the timing program running. The synthetic workload was tuned in this environment. When these workloads are executed with other activities, there are additional overheads that are not included in the task set specifications. These overheads include:

- context switch times and cache effects, which are likely to increase as the task set size increases,
- periodic thread overhead associated with periodically releasing and resetting the computation,
- and interference from interrupts.

The reserved computation time is set to be 1 to 2 milliseconds larger than the pure computation amount to accommodate these overheads.

The second issue is that in some rare cases, a series of interrupts or a large critical region in the kernel may preempt a program and cause it to miss its reserved time and subsequently miss its deadline for the period. To mask these rare instances in Experiments 2 and 3, the 5-percentile and 95-percentile are given. This shows that for the vast majority of reservation periods for these threads, the usage observed is that which is expected based on the reservation.

6.2.2 Client/server synthetic workloads

Most interesting applications are not independent, so it is important to consider experiments that characterize the effect of interactions such as client/server relationships in reserved applications. The experiments described below show that reserved activities can achieve predictable behavior, even when the activities involve coordination between clients and servers.

6.2.2.1 Methodology

The following experiments use the same kind of software environment as described in Section 6.2.1. There is a configuration manager that reads a specification of a task set and then creates the programs for the task set. In addition to the `arith` and `monitor` programs described above, these experiments use the following programs:

- `tsclient` - Creates a periodic thread that invokes a server to compute for some duration of time specified in the invocation. The invocation is performed using the regular Mach IPC mechanism.
- `tsserver` - Services requests sent in from instantiations of the `tsclient` program.
- `rclient` - Creates a periodic thread that invokes a server to compute for some duration of time, but unlike the `tsclient`, the `rclient` uses RT-IPC instead of regular Mach IPC, and the `rclient` sends a reserve to the server so that it can charge the computation time to the client's reserve.

- `rserver` - Services requests sent from instantiations of the `rclient` program. Uses RT-IPC and charges the computation requested by a client to the client's reserve, which is passed as an argument with the invocation.

Experiment 4 is designed to show the processor usage of a client/server pair that has no competition from other programs; this is the base case, showing the desired behavior for the client and server. It uses a task set with an instance of the `tsclient` program using Mach IPC to periodically invoke an instance of `tserver` to perform a computation. A monitor records the usage for later analysis. In this case, the client sends the computation time amount (to be consumed in a tight loop) to the server. And the server computes for that amount of time and returns a result. The parameters for the programs in this task set are given in Table 6-5. The client is periodic and has a reservation associated with it. As long as the computation time requested by the client is smaller than the period, the server with no competition should be able to finish the computation by the end of the period, yielding a fairly constant utilization over time.

Program	#	Program Computation	Program Period	Reserved Computation	Reservation Period
<code>tsclient</code>	1	10 ms	40 ms	10 ms	40 ms
<code>tserver</code>	1	infinite loop	N/A	0 ms	40 ms
<code>monitor</code>	1	N/A	20 ms	1 ms	20 ms

Table 6-5: Experiment 4 Parameters

In Experiment 5, the task set includes competition from unreserved programs as well as the `tsclient`, `tserver`, and `monitor`. This experiment is meant to show how competition for the processor from unreserved activity can interfere with the coordinated activity of a client and server using a typical IPC mechanism.

Program	#	Program Computation	Program Period	Reserved Computation	Reservation Period
<code>rclient</code>	1	8 ms	40 ms	10 ms	40 ms
<code>rserver</code>	1	N/A	N/A	0 ms	40 ms
<code>arith</code>	5	infinite loop	N/A	0 ms	40 ms
<code>monitor</code>	1	N/A	20 ms	2 ms	20 ms

Table 6-6: Experiment 6 Parameters

Experiment 6 is designed to determine whether a client/server pair, using an IPC mechanism integrated with the reservation system in terms of queueing and scheduling, can sustain a predictable, coordinated activity even with competition for the processor. Table 6-6 shows the task set specification for Experiment 6. The `rclient` has a processor reservation, and the `rclient` and `rserver` communicate using RT-IPC. The competition for the processor comes from five `arith` programs which are unreserved.

Program	#	Program Computation	Program Period	Reserved Computation	Reservation Period
<code>rclient1</code>	1	8 ms	40 ms	10 ms	40 ms
<code>rclient2</code>	1	8 ms	50 ms	10 ms	50 ms
<code>rclient3</code>	1	8 ms	60 ms	10 ms	60 ms
<code>rserver</code>	1	N/A	N/A	0 ms	40 ms
<code>arith</code>	5	infinite loop	N/A	0 ms	40 ms
<code>monitor</code>	1	N/A	20 ms	2 ms	20 ms

Table 6-7: Experiment 7 Parameters

Experiment 7 is intended to show whether several reserved clients can execute in a manner that satisfies their timing constraints when using the same server and competing with unreserved, compute-bound programs. The task set, shown in Table 6-7 shows three reserved instances of the `rclient` program with different reservation parameters. There is also an instance of the `rserver` program. The competition comes from five instances of the `arith` program which are unreserved, and there is a reserved `monitor` program.

Program	#	Program Computation	Program Period	Reserved Computation	Reservation Period
<code>rclient</code>	1	8 ms	40 ms	0 ms	40 ms
<code>rclient</code>	1	8 ms	50 ms	10 ms	50 ms
<code>rclient</code>	1	8 ms	60 ms	10 ms	60 ms
<code>rserver</code>	1	N/A	N/A	0 ms	20 ms
<code>arith</code>	5	infinite loop	N/A	0 ms	40 ms
<code>monitor</code>	1	N/A	20 ms	2 ms	20 ms

Table 6-8: Experiment 8 Parameters

Finally, Experiment 8 is designed to determine whether reserved clients can meet their timing constraints if there is an unreserved client that is using the same server. As shown in Table 6-8, the task set for Experiment 8 contains one `rclient` program with no reservation and two `rclient` programs with reservations. There is an `rserver` and five competing `arith` programs which are unreserved. A monitor is also included in the task set.

6.2.2.2 Results

The results from Experiment 4, shown in Figure 6-6, illustrate the processor usage pattern of the periodic client and its server. The x-axis is time over the duration of the test measured in seconds, and the y-axis is processor utilization. The usage measurements for both the client and the server are taken from the corresponding reserves. The client has a reserve that it charges for its own computation, and the server has a reserve that it charges against when performing an operation for a client. For each of these reserves, the monitor records the computation time used in each reservation period. Those computation times are then normalized with respect to the length of the corresponding reservation periods and plotted as constant for the duration of each reservation period.

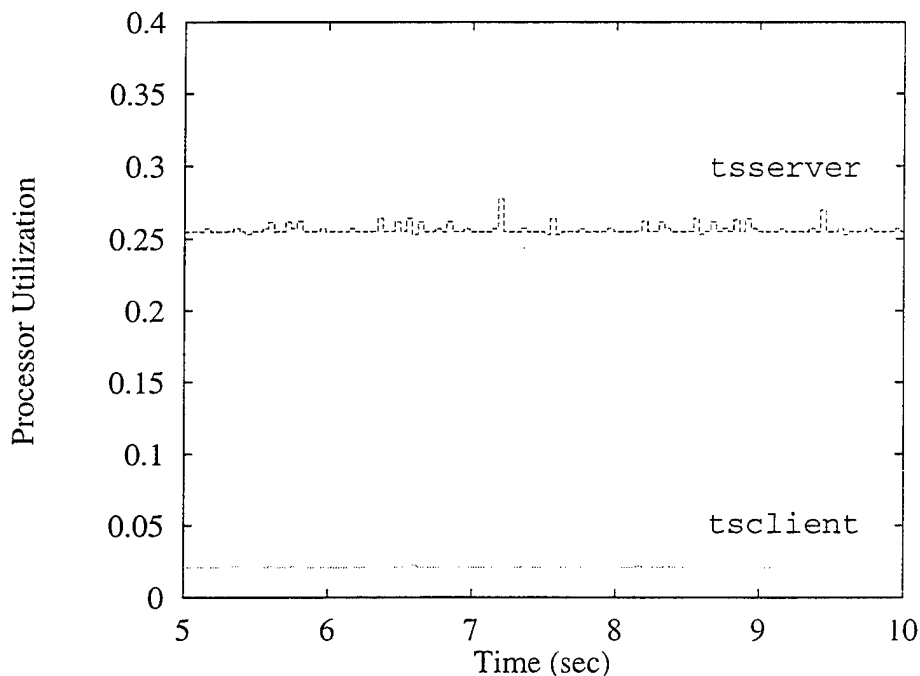


Figure 6-6: Experiment 4 Results

Figure 6-6 shows that the processor utilization charged to the server's reserve is fairly constant over the duration of the test. The average is 0.256 with a 5-percentile of 0.255 and a 95-percentile of 0.264 for the measurements graphed in the figure. The normalized charges to the client reserve average only 0.0210; the 5-percentile is 0.0207 and the 95-percentile is 0.0213. In this setup, the server is doing most of the work while the client does no

work other than sending off requests and receiving replies. Since there is no competition and the client makes the same request in every period, the utilization is fairly constant over the duration of the test.

Figure 6-7 shows the results of Experiment 5 where the same client/server pair has competition for the processor from unreserved activities. As before, the x-axis is time measured in seconds, and the y-axis is processor utilization. In this case, the client and server do not have constant utilization numbers over each reservation period. For the measurements shown in the graph, the server has an average utilization of 0.191, which is significantly lower than the desired level. The 5-percentile for the server is 0, and the 95-percentile is 0.262. The client has an average utilization of 0.0117 across the periods shown in the graph: its 5-percentile is 0 and its 95-percentile is 0.0158. With the client and the server recording 0 utilization in a significant number of periods in a row, it is clear that the client/server combination is not achieving the desired behavior.

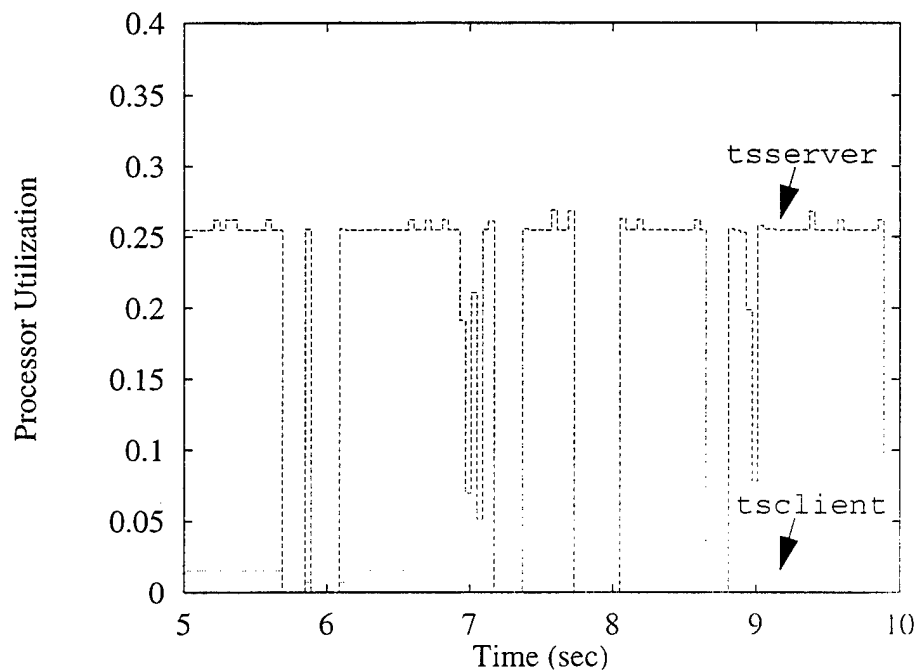


Figure 6-7: Experiment 5 Results

The competition from unreserved programs interferes with the execution of the server, and completely locks out the server for up to 100 to 200 ms at a time. During these periods, there is no usage recorded by either the server or the client, since the client cannot make progress without the server making progress. The usage for both the client and the server falls to zero for several reservation periods. This kind of behavior is clearly undesirable since many instances of the computation cannot take place, and the deadline is missed each time.

The results of Experiment 6 (Figure 6-8) show that when a client and server use an IPC mechanism that is integrated with the reservation scheduling policy, in this case a version of RT-IPC extended to work with the reservation scheduling policy, the combined client/server activity is quite predictable. The RT-IPC mechanism propagates the client's reserve to the server and supports a server that charges the computation time of each client to the client's reserve. So most of the computation in this experiment is being charged to the client (as it should be) instead of to the server (as in the previous case). The utilization charged to the client's reserve averages 0.223 with a 5-percentile of 0.222 and a 95-percentile of 0.228. The server utilization for the graphed intervals is 0.0114 on average; the 5-percentile is 0.0112 and the 95-percentile is 0.0117. These numbers indicate very predictable performance for these programs over time.

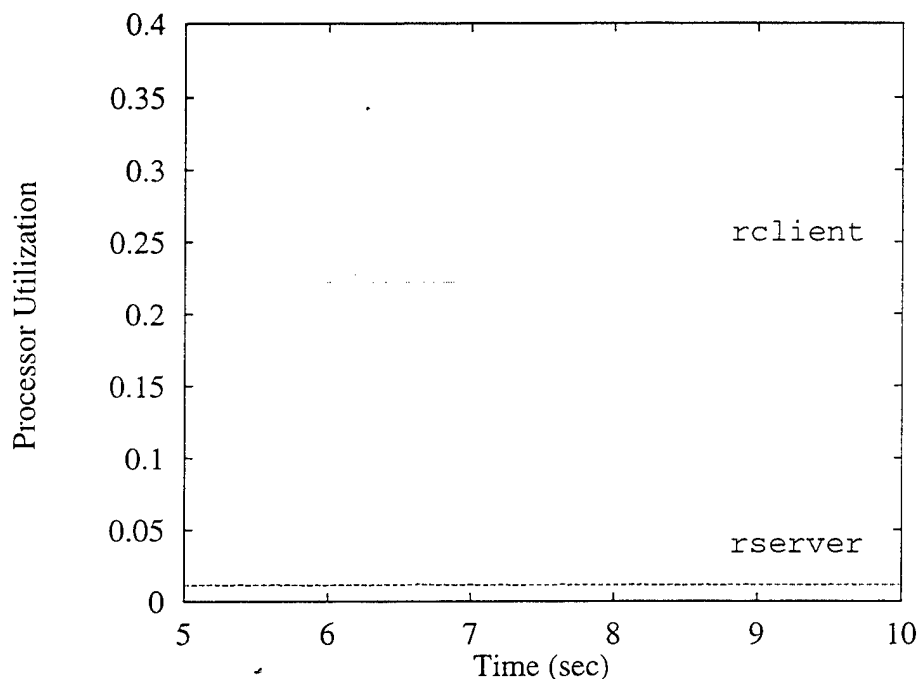


Figure 6-8: Experiment 6 Results

Figure 6-9 shows the results from Experiment 7. These results show that even when several reserved clients are using the same server, they can all meet their periodic timing constraints (subject to the admission control policy). This is true in spite of the presence of competition from unreserved arith programs.

Experiment 7 shows the usage charged to the reserves of the three reserved clients. The client with the 8ms/40ms synthetic computation has an average utilization of 0.223 with a 5-percentile of 0.221 and a 95-percentile of 0.232. The client with the 8ms/50ms computation has an average utilization of 0.178, a 5-percentile of 0.176 and a 95-percentile of 0.184. And the client with the 8ms/60ms computation gets an average utilization of 0.150; the 5-percentile is 0.147 and the 95-percentile is 0.157. These numbers indicate fairly tight distributions

around the averages for these applications, even though they are competing for the server and even though there are additional unreserved programs competing for the processor as well. The servers computation time seems erratic, and there are two reasons: it has a small reservation period (which just determines the usage measurement period), and its clients all have different periods and request different computations at different rates.

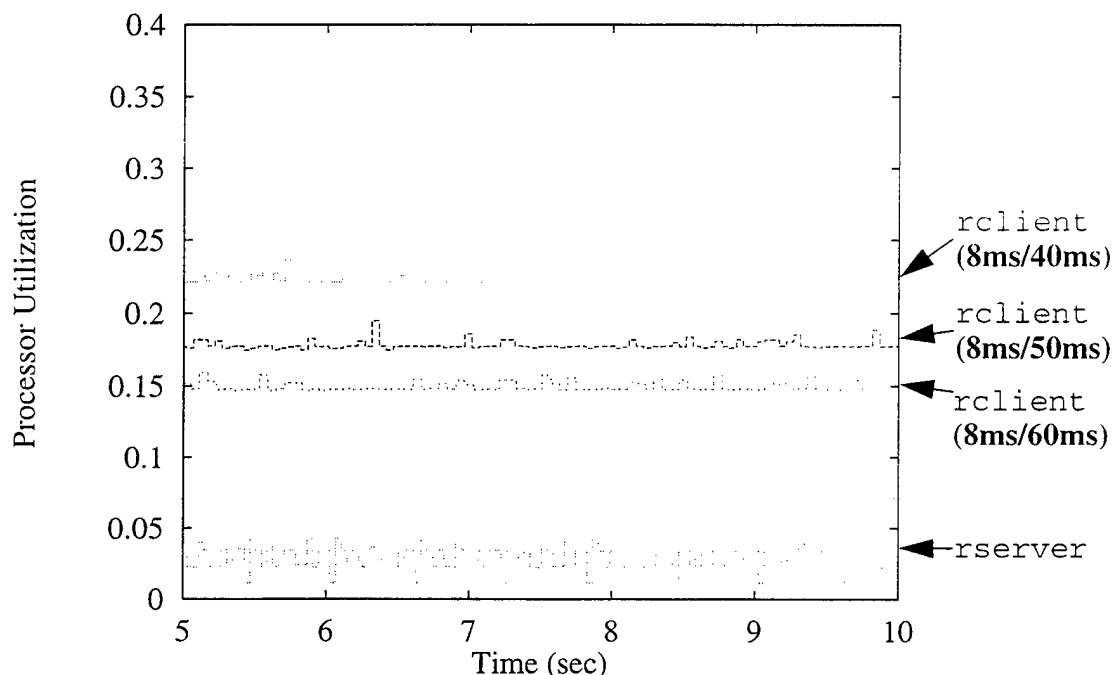


Figure 6-9: Experiment 7 Results

Finally, the results of Experiment 8 appear in Figure 6-10. These results show that in the case where reserved clients compete with an unreserved client for a single server, the reserved clients are still able to satisfy their timing constraints.

The reserved client with the 8ms/50ms computation has an average utilization of 0.181. It has a 5-percentile of 0.176 and a 95-percentile of 0.187. The reserved client with the 8ms/60ms computation has an average utilization of 0.151 with a 5-percentile of 0.148 and a 95-percentile of 0.158. Thus, these reserved programs are able to get the processor time they have reserved. As the graph shows, the unreserved client manages to complete its computation during some of its periods, but not in others. So the usage function goes back and forth between getting about 0.22 utilization in the periods where the computation is completed and getting 0 utilization in the periods where it does not get to complete the computation. The average utilization for this unreserved client is 0.131; the 5-percentile is 0.0059 and the 95-percentile is 0.222. This of course confirms that the dispersion of the utilization measurements for this unreserved client is large.

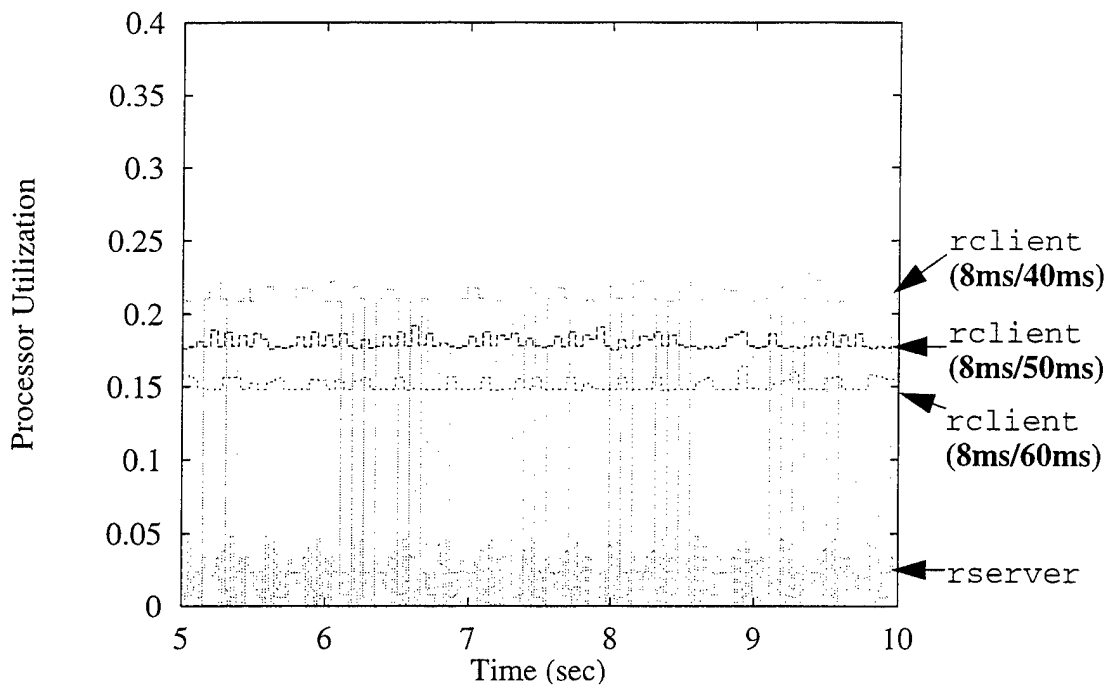


Figure 6-10: Experiment 8 Results

6.2.2.3 Analysis

The experiments with unreserved and reserved client/server pairs demonstrate the importance of doing reserve propagation properly between client and server when the server is designed to use the client's reserve. Experiment 4 shows the baseline behavior for the client/server pair with a periodic client driving the timing of the activity.

Experiment 5 demonstrates the problem that can occur when the reserve propagation is not handled properly. In this experiment, the client allocates a reserve and passes it to the server, which then uses it to charge the client's service time. However, this client/server pair does not use the "priority" inheritance mechanism to ensure that the server takes on the "priority" of the client as soon as the RPC is enqueued in the servers input queue. With competition from unreserved activities, this lack of "priority" inheritance results in many missed deadlines for the client/server activity.

The results of Experiment 6 show that the proper periodic behavior of the client/server pair can be restored by using the "priority" inheritance mechanism. Priority inheritance makes sure that the competing unreserved activities do not interfere with the server as it attempts to read the request from its input queue and switch to the client's reserve.

The last two experiments demonstrate that the reserve propagation mechanism, which includes "priority" inheritance and the server binding to the client's reserve, works properly even when there are multiple reserved clients or there are unreserved clients in addition to

reserved clients. Experiment 7 demonstrates that with three reserved clients (all with different computation times and timing constraints), the server can service them all in time to make their deadlines. This is true even though there is competition from unreserved activities, which can exploit any lapses in an incorrectly implemented reserve propagation mechanism and cause delays in the client/server activities.

Experiment 8 shows that when two reserved clients and one unreserved client share a server in an environment with competition from other unreserved activities, the reserved clients will always meet their deadlines. In this case the reserved clients meet their deadlines even though other unreserved competitors sometimes delay the unreserved client. This experiment further tests the integrity of the reserve propagation mechanism by making sure that the “priority” of the unreserved client is not propagated to the server at the wrong time, causing the server to appear unreserved and resulting in interference from the unreserved competitors.

6.2.3 QTPlay/X Server

Experiments using task sets with synthetic workloads provide evidence that the reservation system can support the predictable execution of real-time programs. However, experiments with real applications that use the reservation system to meet timing constraints provide stronger evidence of the usefulness of the reservation system in real-world situations. The experiments described in this section use a video player that has been modified to use processor reserves and a version of the X Server that has been modified to support reserves.

6.2.3.1 Methodology

These experiments use the QuickTime video player, called QTPlay, and the reserved X Server, both of which were described in the previous chapter. Since these programs are described in detail elsewhere, the description here is brief.

The QTPlay application prefetches a short video clip into main memory and repeatedly displays that clip to avoid interaction with the file system and disk (which are not reserved in this system) during the experiments. It allocates a reserve during initialization based on command-line arguments and then starts playing the video. For each frame, the player records in a buffer the start time and end time for the frame processing, and at the end of the experiment these data are written to a file on the disk for later analysis.

When QTPlay connects to the X Server, it passes a reference to its reserve for the X Server to use when performing frame display operations. The X Server was modified to order requests based on reservation information and to charge the computation time for each operation to the appropriate client’s reserve.

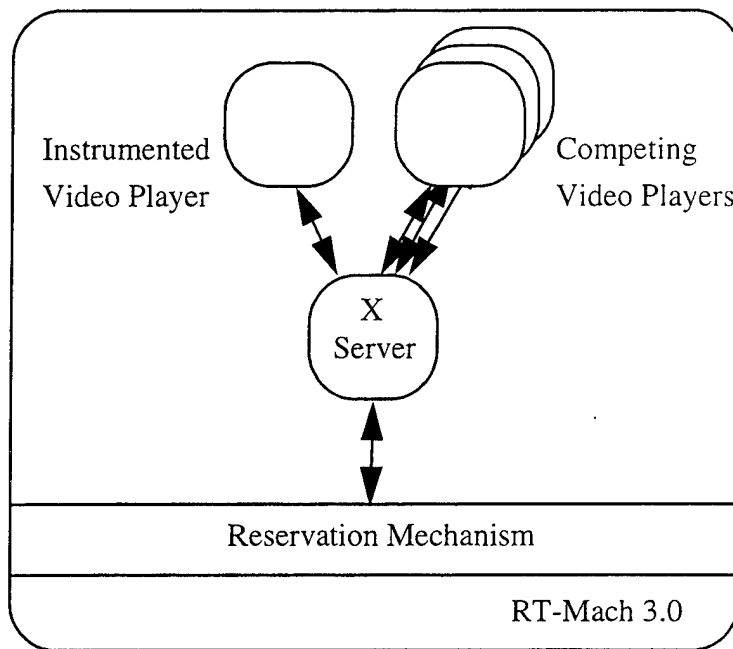


Figure 6-11: Software Configuration

Figure 6-11 shows the basic software structure that is used for all of the experiments in this section. There is an instrumented QTPPlay application which may or may not have a reservation and which records timestamps for each frame at the beginning of the frame display computation and then again at the completion of frame display. Other instances of QTPPlay may compete with this instrumented player. These are unreserved and continuously display frames as fast as possible (providing the maximum competition).

QTPPlay can display frames at a particular period or in a continuous loop, and in all of the experiments below, the frame resolution is 160x120 pixels with 8 bits/pixel. The timing is specified by command-line arguments.

Experiment 9 is designed to illustrate the usage pattern for QTPPlay with no competition for the processor. The parameters for QTPPlay are given in Table 6-9. The period is 33 ms, which corresponds to a frame rate of 30 frames/second.

Program	#	Mode	Period	Reserved Computation	Reservation Period
QTPPlay	1	Periodic	33 ms	0 ms	0 ms

Table 6-9: Experiment 9 Parameters

Experiment 10 is intended to show what can happen when QTPlay is executed under a time-sharing scheduler with a competing instance of the QTPlay program. The parameters for this experiment appear in Table 6-10. The QTPlay instance listed in the first row of the table is instrumented to provide timing information and the other just competes for the resources for displaying frames by continuously displaying frames as fast as possible.

Program	#	Mode	Period	Reserved Computation	Reservation Period
QTPlay	1	Periodic	33 ms	0 ms	0 ms
QTPlay	1	Continuous	N/A	0 ms	0 ms

Table 6-10: Experiment 10 Parameters

Experiment 11 is designed to show how well an instance of QTPlay with a reservation can coordinate with the reserved X Server to achieve a constant playback rate for frames. Table 6-11 gives the parameters for the experiment. The instrumented QTPlay application has a reservation and competition from one other unreserved QTPlay instance.

Program	#	Mode	Period	Reserved Computation	Reservation Period
QTPlay	1	Periodic	33 ms	14 ms	33 ms
QTPlay	1	Continuous	N/A	0 ms	0 ms

Table 6-11: Experiment 11 Parameters

Experiment 12 is similar to Experiment 10 in that it explores the behavior of an unreserved QTPlay with competition from 3 unreserved QTPlay instances rather than just one. For completeness, the parameters appear in Table 6-12.

Program	#	Mode	Period	Reserved Computation	Reservation Period
QTPlay	1	Periodic	33 ms	0 ms	0 ms
QTPlay	3	Continuous	N/A	0 ms	0 ms

Table 6-12: Experiment 12 Parameters

Experiment 13 is similar to Experiment 11; it looks at the behavior of a reserved QTPlay instance with three competing QTPlay instances. The parameters are given in Table 6-13. These last two experiments look at the behavior of unreserved and reserved QTplay applications under adverse conditions (intense competition from multiple unreserved X clients).

Program	#	Mode	Period	Reserved Computation	Reservation Period
QTPlay	1	Periodic	33 ms	14 ms	33 ms
QTPlay	3	Continuous	N/A	0 ms	0 ms

Table 6-13: Experiment 13 Parameters

6.2.3.2 Results

The results for Experiment 9 illustrate the timing behavior of a QTPlay application with no competition. Figure 6-12 shows these results. For each frame, the player records the starting time and ending time. The x-axis is the frame number, counting frames starting at the 200th frame through to the 400th. For each frame value, the difference between the start time and end time is computed, and the y-axis is this frame delay measured in milliseconds.

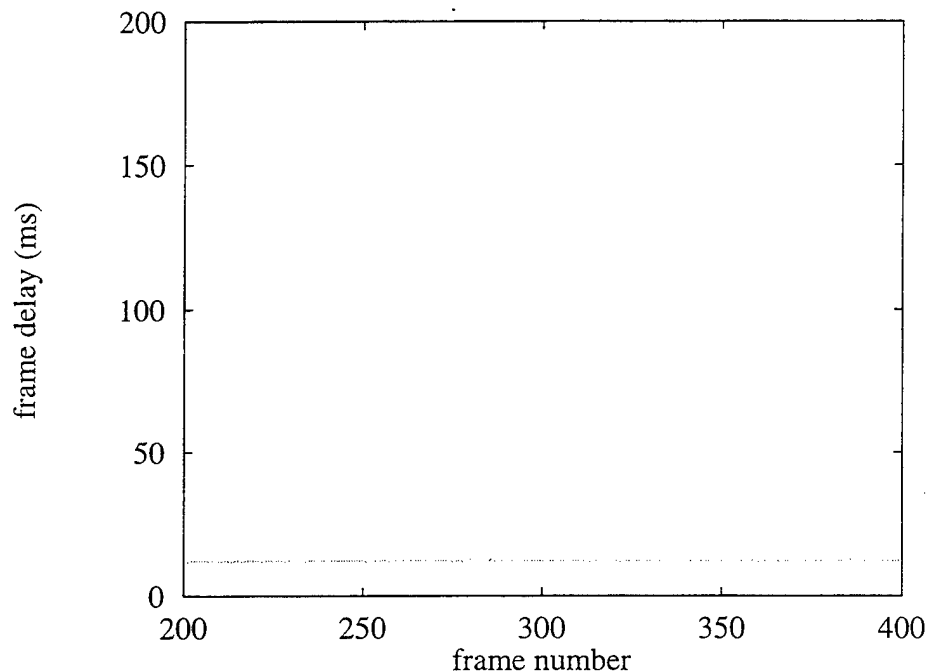


Figure 6-12: Experiment 9 Results

In the figure, the frame delay averages 12.1 ms with a 5-percentile of 11.8 and a 95-percentile of 12.6. This indicates that the software took an average of about 12 ms to perform all the computations necessary to display a frame when there was no competition for resources.

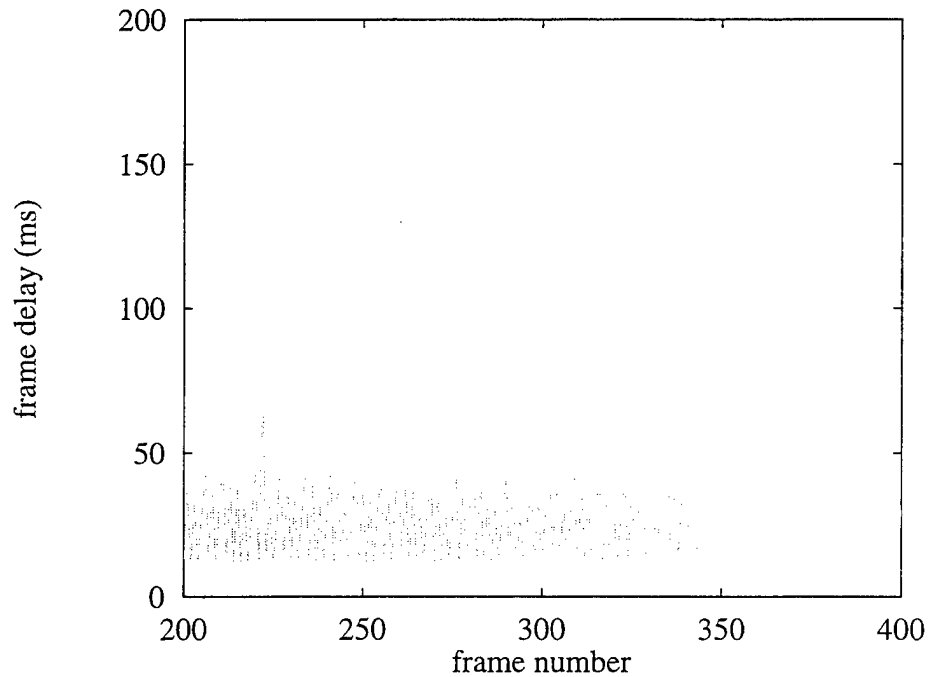


Figure 6-13: Experiment 10 Results

The results from Experiment 10 show a slightly different picture in Figure 6-13. Again, the x-axis is frame number, and the y-axis is frame delay measured in ms using the same method. With time-sharing scheduling and one competing QTPlay, the instrumented QTPlay sees quite a bit of interference in its frame delay time. The frame delay is much more variable. The average delay is 25.8 ms with a 5-percentile of 11.2 and a 95-percentile of 45.0.

In the results from Experiment 11, the instrumented QTPlay has a reservation, and its frame delay is much less variable even with competition from one unreserved QTPlay instance. Figure 6-14 shows the timing behavior. As before, the x-axis is frame number; the y-axis is frame delay in milliseconds.

The frame delay still has a bit of variation, but it is much less variable than the case where the QTPlay application is unreserved. The average delay is 19.4 ms with a 5-percentile of 14.7 and a 95-percentile of 24.3. So QTPlay is almost always able to display each frame within its 33 ms period.

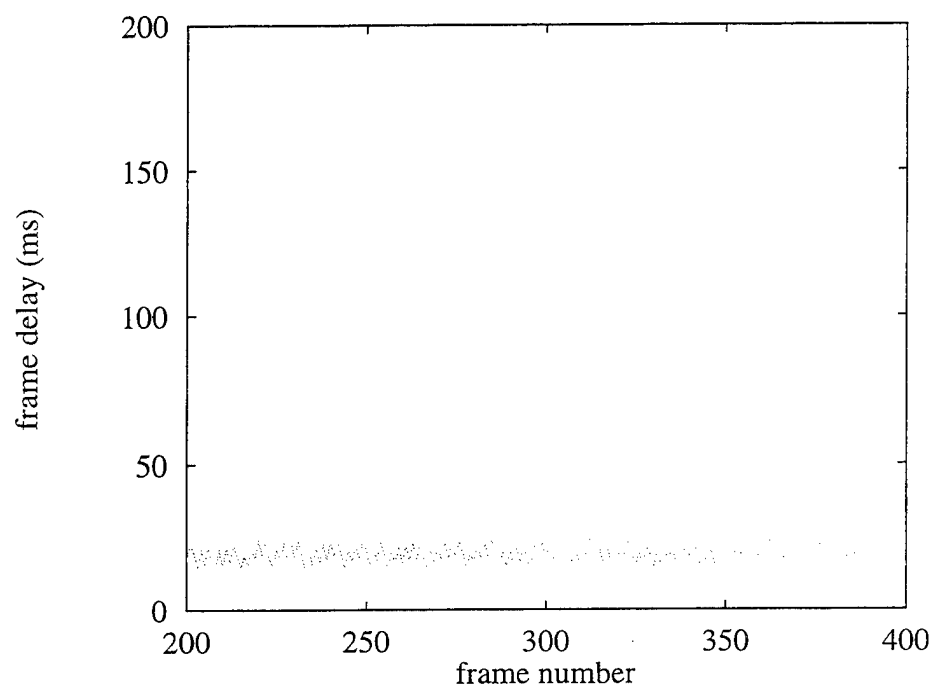


Figure 6-14: Experiment 11 Results

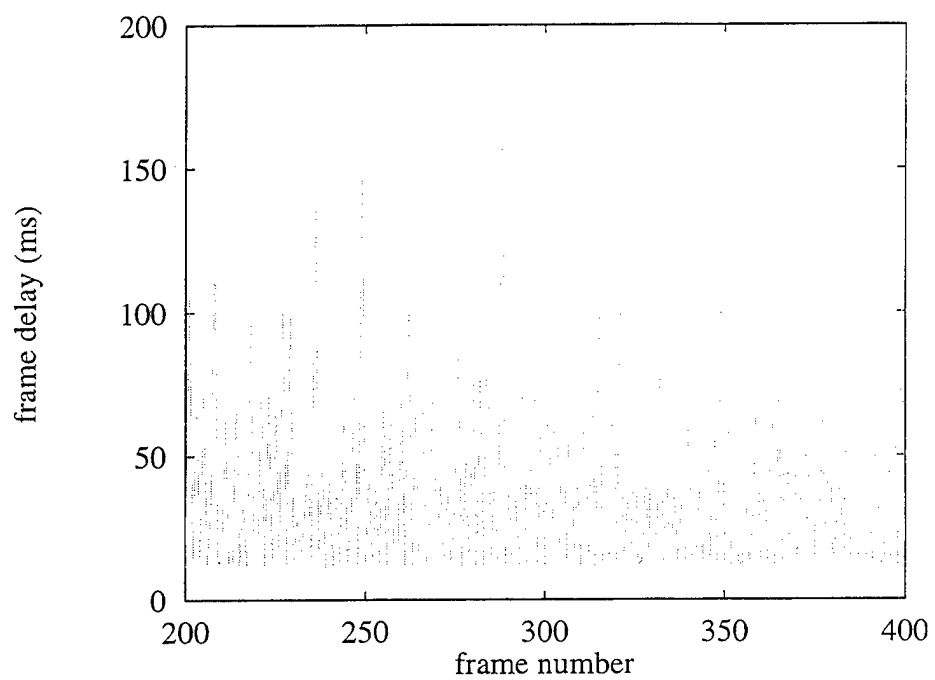


Figure 6-15: Experiment 12 Results

The results from Experiment 12 show how much the frame delay variation can be for an unreserved QTPlay instance that has three unreserved QTPlay applications competing to display frames. Figure 6-15 shows these results. As the figure shows, the variation in frame delay is quite large. The 5-percentile for the frame delay is 11.5 ms, and the 95-percentile is 107 ms with an average frame delay of 40.3 ms. A delay of 150 ms (which does not show up in the 95-percentile number but occurs a number of times during the test) or even 100 ms in a sequence of video frames is clearly noticeable to the human eye. Frame rates of 15 frame/second or more are required to sustain the illusion of smooth motion. This implies that with delays above 66 ms or so, the illusion of smooth motion may be destroyed.

In contrast, the results of Experiment 13, shown in Figure 6-16, demonstrate how well the reservation system can control the variability in frame delay for the reserved QTPlay application, even with much competition from unreserved instances of the QTPlay program. The frame delay in the figure is somewhat variable, but the variation is much less than in the case of the unreserved QTPlay with three competitors. The 5-percentile is 13.4 ms and the 95-percentile is 34.2 ms with an average of 20.7 ms. This is well below the target of the 60 ms period necessary for smooth-looking motion.

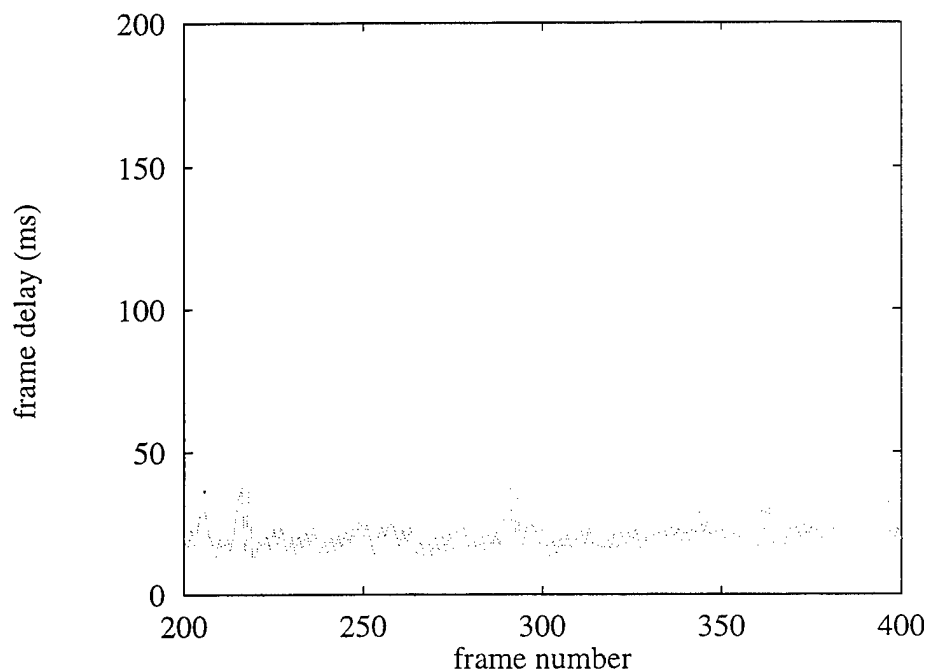


Figure 6-16: Experiment 13 Results

6.2.3.3 Analysis

The QTPlay/X Server experiments show that even with real applications like a Quick-Time video player and the X server, the reservation system can provide predictable perfor-

mance for real-time programs that must meet timing constraints to achieve satisfactory performance.

Experiment 9 shows the baseline behavior that is desired for the video player. This experiment has no competing activity, so there is no contention for resources and the behavior is very regular. Experiment 10 has a competing video player in addition to the instrumented video player, and with time-sharing scheduling, this competitor causes some scheduling delay in the instrumented player. In Experiment 12, there are three competitors, and the interference to the instrumented player is very bad. The player frequently has frame delays of 60 to 100 ms and even as high as 150 ms. These kinds of delays are clearly noticeable to a human observer of the video stream.

In Experiment 11, the instrumented video player is reserved with one competitor, and although there is a little fluctuation in the frame delay, it is limited to 24.3 for the 95-percentile. In Experiment 13, the reserved video player has competition from three unreserved video players. The frame delays show a little more variability, but are still limited to a 95-percentile of 34.2 ms compared to the 100 and even 150 ms delays experienced with time-sharing scheduling.

The question of why the reserved QTPlay/X Server combination suffered any delay arises. The reason is that the X Server was not implemented from scratch to use reserves. The extensions to allow it to use reserves did not completely restructure the request input queue, in particular. So the server reads requests from its input queue, orders them internally, and then performs the operations. If server's client interface code were completely rewritten to support reserves, the behavior would be comparable to that of the client/server synthetic benchmarks where the servers were designed from scratch.

6.2.4 mpeg_play/X Server

In addition to the QuickTime video player, a version of the `mpeg_play` decoder was modified to use reserves and coordinate with the reserved X Server. This decoder uses some simple usage measurement and adaptation techniques to tune the reservation parameters and timing parameters based on changing system conditions.

6.2.4.1 Methodology

The `mpeg_play` modifications were described in detail in the previous chapter, so the description here is brief. The player prefetches a video clip into memory to avoid interference in the file system. It requests a processor reservation and passes the reference to its reserve to the X Server. While it is executing, the decoder keeps track of its resource usage and timing characteristics, and it makes adjustments to the reservation parameters and/or period of the program based on usage.

Experiment 14 is designed to determine whether the `mpeg_play` decoder can successfully modify its reservation parameters and/or behavior based on existing conditions. The decoder starts executing with a reservation that is too small for its computation time. Competition is then introduced in the form of reserved and unreserved activities.

6.2.4.2 Results

The behavior of the `mpeg_play` application under the conditions of Experiment 14 is illustrated in Figure 6-17. The decoder is able to tune its reservation parameters based on run-time information and stabilize its own behavior.

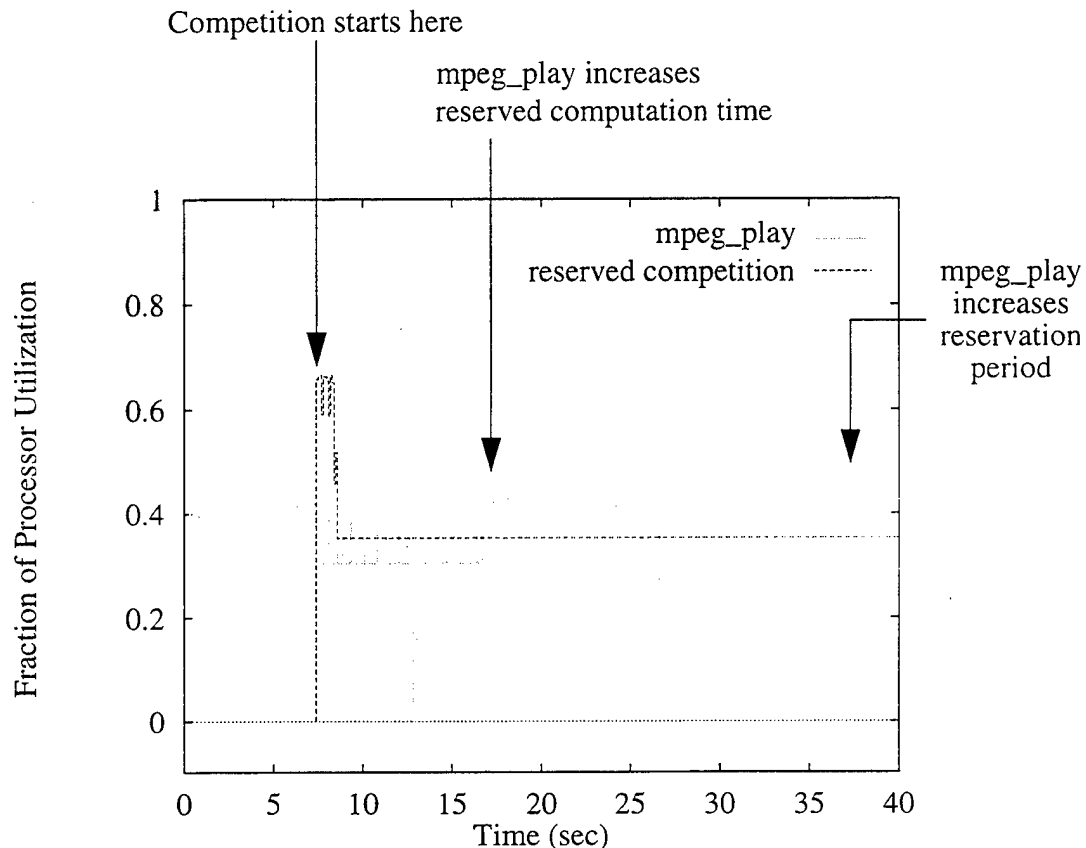


Figure 6-17: Experiment 14 Results

Figure 6-17 shows the processor utilization of the `mpeg_play` decoder over a period of 40 seconds. The x-axis is time in seconds, and the y-axis is processor utilization. During the first 7 seconds, `mpeg_play` averages about 40% of the processor, even though its reservation is only 30 ms every 100 ms. Since there is no competition, it consumes 40%. At about 7 seconds into the experiment, a reserved program (shown in the graph) and several unreserved programs (not shown) are introduced. The usage of `mpeg_play` immediately drops to its reserved level of 30% of the processor. This is not enough to sustain its previous frame rate, so some frames are dropped.

The reserved activity, which had a usage spike at the time it started, settles down to a constant 37%. The spike occurs since the time-sharing algorithm initially allows the new program to get more cycles than its reservation. After consuming a large percentage of the

processor, however, the reserved activity no longer gets additional cycles from the time-sharing algorithm, and the usage flattens.

After several seconds, `mpeg_play` realizes its frame rate has fallen and attempts to increase its reservation. At about 17 seconds into the experiment, the decoder increases its reservation to 41 ms reserved every 100 ms, and its frame rate increases accordingly. Again at about 37 seconds into the experiment, the decoder changes its reserved computation time to 47 ms and its reservation period to 111 ms to fine-tune the reservation even further.

6.2.4.3 Analysis

This experiment demonstrates that an application can adjust its reservation parameters and adapt its behavior based on the usage information from the reservation mechanism. In this case, the initial reservation of the `mpeg_play` application did not have to be accurate since the application automatically adjusted the reservation levels based on usage measurements.

6.2.5 Protocol processing workloads

The experiments in this section explore the real-time behavior of the socket library (called `libsockets`) protocol processing. The behavior obtained by an application using the socket library is compared to the behavior using the UX Server's socket service under both time-sharing scheduling and reserves.

6.2.5.1 Methodology

The `libsockets` experiments use the same software environment as the experiments with independent tasks and client/server workloads. There is a configuration manager that reads the task set specification and creates the specified programs. In addition to the workload programs introduced so far, these experiments use the following programs:

- `stdio` - Creates a periodic thread that calls a sequence of file operations.
- `udps` - Creates a periodic thread that sends some number of packets (specified in the program computation field) in each period. This program uses the UX Server to send packets.
- `udpls` - Creates a periodic thread that sends some number of packets in each period. This program uses `libsockets` to send the packets rather than interacting with the UX Server.
- `udpr` - Creates a periodic thread that receives some number of packets in each period using the UX Server to receive the packets.
- `udplr` - Creates a periodic thread that receives some number of packets in each period using `libsockets`.

Experiment 15 is designed to show how a packet-sending activity can be disturbed by competition from a combination of compute-intensive and I/O-intensive activities, especially when all of those activities use system services which interact with each other as they do in the UX Server. The task set for this experiment is given in Table 6-14. It shows two udp senders (udps) with slightly different workloads. The competition for this experiment (as for the next three experiments) consists of five compute-intensive arith programs and five I/O-intensive stdio programs. In this experiment, the packets sent by the udps programs are received on a remote machine by yet another program that records a timestamp when the each packet arrives. This program buffers the timestamps and dumps them out at the end. The timestamp data can be used to judge whether the packet senders were able to send their packets out as desired or not.

Program	#	Program Computation	Program Period	Reserved Computation	Reservation Period
udps (A)	1	4 pkt	40 ms	0 ms	40 ms
udps (B)	1	2 pkt	40 ms	0 ms	40 ms
arith	5	various	various	0 ms	40 ms
stdio	5	various	various	0 ms	40 ms
monitor	1	N/A	20 ms	1 ms	20 ms

Table 6-14: Experiment 15 Parameters

Experiment 16 is designed to determine whether the packet-sending applications can send their packets on time when using libsockets for their network protocol processing. The parameters are given in Table 6-15. This experiment differs from Experiment 15 in that the UDP packet senders use libsockets instead of the UX Server and they have reservations instead of being scheduled by the time-sharing scheduler. The competition is the same: five compute-intensive programs and five I/O-intensive programs.

Program	#	Program Computation	Program Period	Reserved Computation	Reservation Period
udpls (A)	1	4 pkt	40 ms	10 ms	40 ms
udpls (B)	1	2 pkt	40 ms	6 ms	40 ms
arith	5	various	various	0 ms	40 ms
stdio	5	various	various	0 ms	40 ms
monitor	1	N/A	20 ms	1 ms	20 ms

Table 6-15: Experiment 16 Parameters

The purpose Experiment 17 is to determine whether a UDP packet-receiving program that attempts to receive a number of packets periodically can meet that objective. The `udpr` program attempts to receive some number of packets in each period. This program receives packets through the UX Server and runs without a reservation. The competing activities are identical to the previous two experiments. Table 6-16 presents the parameters for this experiment.

Program	#	Program Computation	Program Period	Reserved Computation	Reservation Period
udpr (A)	1	4 pkt	40 ms	0 ms	40 ms
udpr (B)	1	3 pkt	40 ms	0 ms	40 ms
arith	5	various	various	0 ms	40 ms
stdio	5	various	various	0 ms	40 ms
monitor	1	N/A	20 ms	1 ms	20 ms

Table 6-16: Experiment 17 Parameters

Experiment 18 is designed to determine whether a reserved packet receiver that uses `libsockets` can predictably execute periodically to receive a number of packets. The `udplr` program is a periodic packet receiver that uses `libsockets` instead of the UX server. This experiment includes the same competition from compute-intensive and I/O intensive tasks as the other experiments in this section. The parameters are given in Table 6-17.

Program	#	Program Computation	Program Period	Reserved Computation	Reservation Period
udplr (A)	1	4 pkt	40 ms	0 ms	40 ms
udplr (B)	1	3 pkt	40 ms	0 ms	40 ms
arith	5	various	various	0 ms	40 ms
stdio	5	various	various	0 ms	40 ms
monitor	1	N/A	20 ms	1 ms	20 ms

Table 6-17: Experiment 18 Parameters

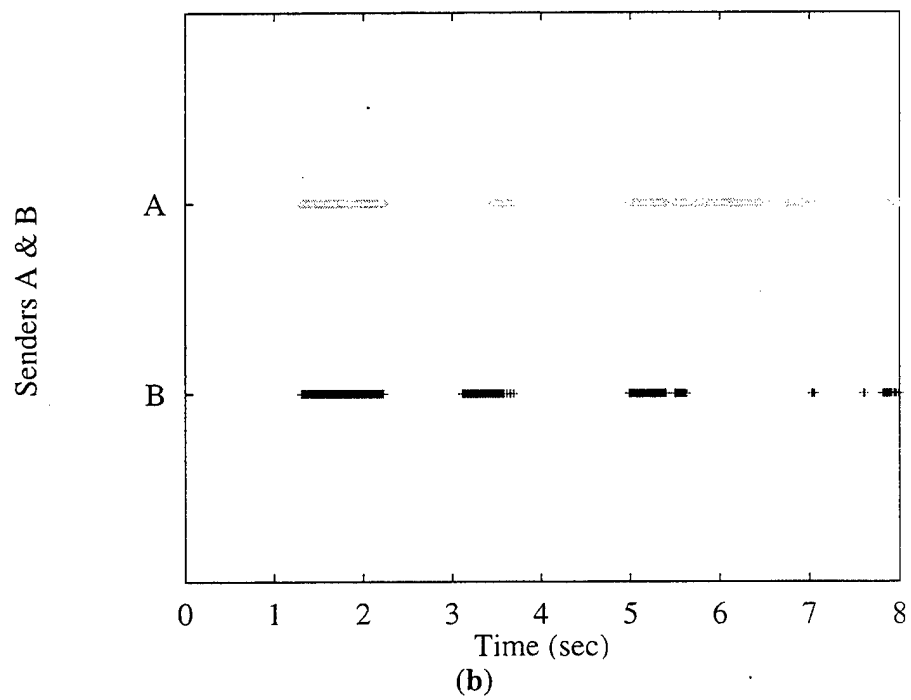
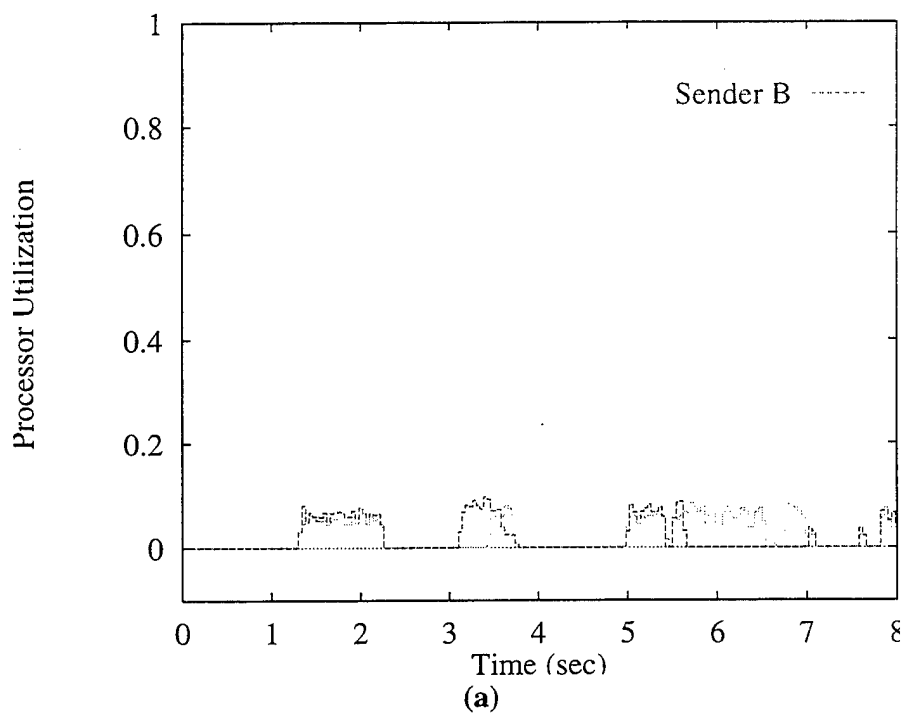


Figure 6-18: Experiment 15 Results

6.2.5.2 Results

The results of Experiment 15 are shown in Figure 6-18. Part (a) of the figure shows the processor utilization over time for the two packet senders, which use the UX Server implementation of sockets and run without reservations. The x-axis is time in seconds, and the y-axis is processor utilization. The two programs, denoted "Sender A" and "Sender B", show a very erratic usage pattern. Frequently, the usage drops to zero for over 1 second. The average utilization for Sender A is 0.023 with a 5-percentile of 0 and a 95-percentile of 0.0741. The average utilization for Sender B is 0.0193 with a 5-percentile of 0 and a 95-percentile of 0.0796. The dispersion is clearly substantial for these applications, and that dispersion in utilization achieved translates directly into missed deadlines.

Figure 6-18(b) shows the record of timestamps that were received by a remote receiver for both senders. The x-axis in the graph is time in seconds, and there are two horizontal lines, one for Sender A and the other for Sender B. A mark on the line corresponding to Sender A at a particular time indicates that a packet arrived at that time and likewise for Sender B. This graph shows that the packets were received on the remote host sporadically. The pattern of packet receptions corresponds closely with the pattern of usage seen in part (a) of the figure. The largest gaps between received packets were 1.28 seconds and 1.22 seconds for Sender A and 1.93 seconds and 1.40 seconds for Sender B. These senders are attempting to send packets every 40 ms, so clearly they are not able to schedule the message sending activity as desired.

The results of Experiment 16 appear in Figure 6-19. In this case, the senders have reservations and use the `libsockets` library to avoid depending on the UX Server for networking. The graph shown in part (a) of the figure shows time in seconds on the x-axis and processor utilization on the y-axis. The processor utilization for both of the senders is very regular, indicating that in each reservation period, the programs were able to send the packets they were supposed to send. There are no long intervals of zero usage as in the previous case. The average utilization of Sender A is 0.103 with a 5-percentile of 0.101 and a 95-percentile of 0.110 indicating a very tight distribution around the average. Likewise for Sender B, the average utilization is 0.0591 and the 5-percentile is 0.0582 with a 95-percentile of 0.0624. This is also a tight distribution.

The graph in Figure 6-19(b) supports the conclusion that the senders in this experiment are able to send their packets very regularly in each period. As before, the x-axis is time in seconds and two horizontal lines indicate the timestamps for packets received from the two senders. A point on the line corresponding to Sender A represents a packet that was received at the associated time. In this experiment, packets are received very regularly from each sender. There are no significant gaps in the reception pattern. The maximum gaps for packets received from Sender A are 0.0519 seconds and 0.0423 seconds, and the maximum gaps for packets received from Sender B are 0.0438 seconds and 0.0435 seconds. So the conclusion is that the combination of `libsockets` with reserved resources yields predictable behavior for packet senders.

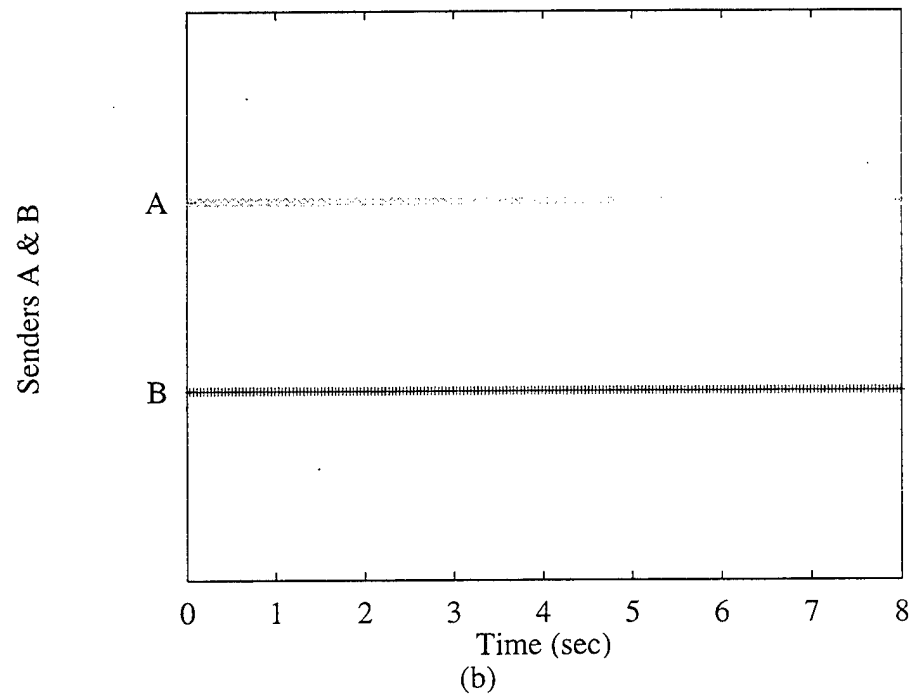
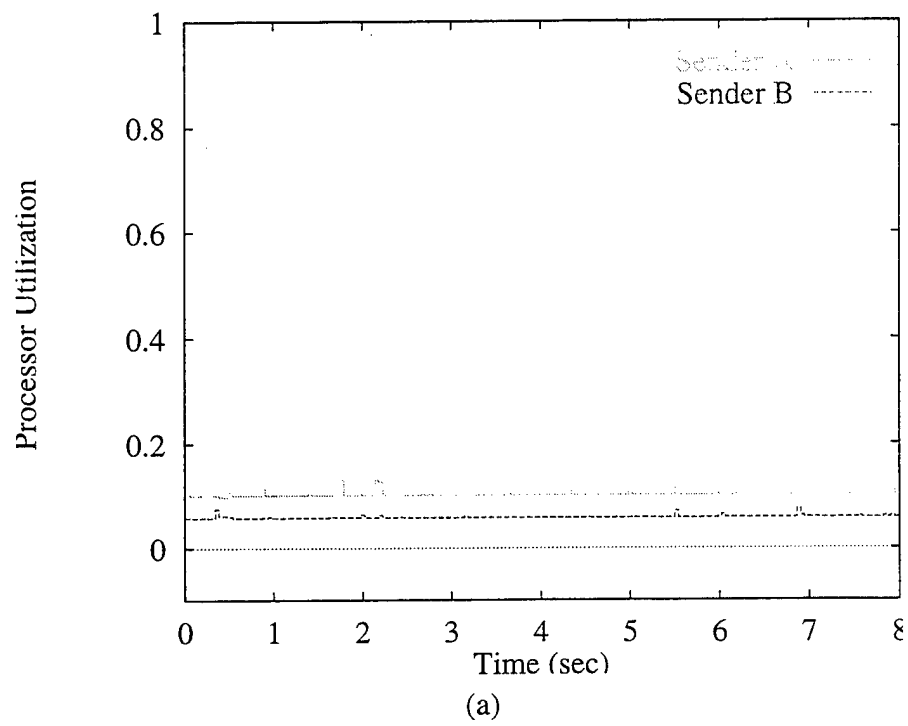


Figure 6-19: Experiment 16 Results

Figure 6-20 shows the results of Experiment 17. In this case, a remote sender periodically sends packets to the two receivers described in the task set. In this experiment, the two receivers are unreserved and use the UX Server's implementation of sockets. The graph has time in seconds on the x-axis and processor utilization on the y-axis. The behavior is very erratic. The remote host sends packets periodically, but the receiver is not always able to run long enough to receive the packets. Receiver A has an average utilization of 0.0134 with a 5-percentile of 0 and a 95-percentile of 0.0617. This is not the kind of timely behavior desired in the packet receiver. Receiver B has an average utilization of 0.0220 with a 5-percentile of 0 and a 95-percentile of 0.0726. Again, the dispersion is significant. The usage for both receivers frequently drops to zero, indicating that the packets are being dropped for significant periods of time.

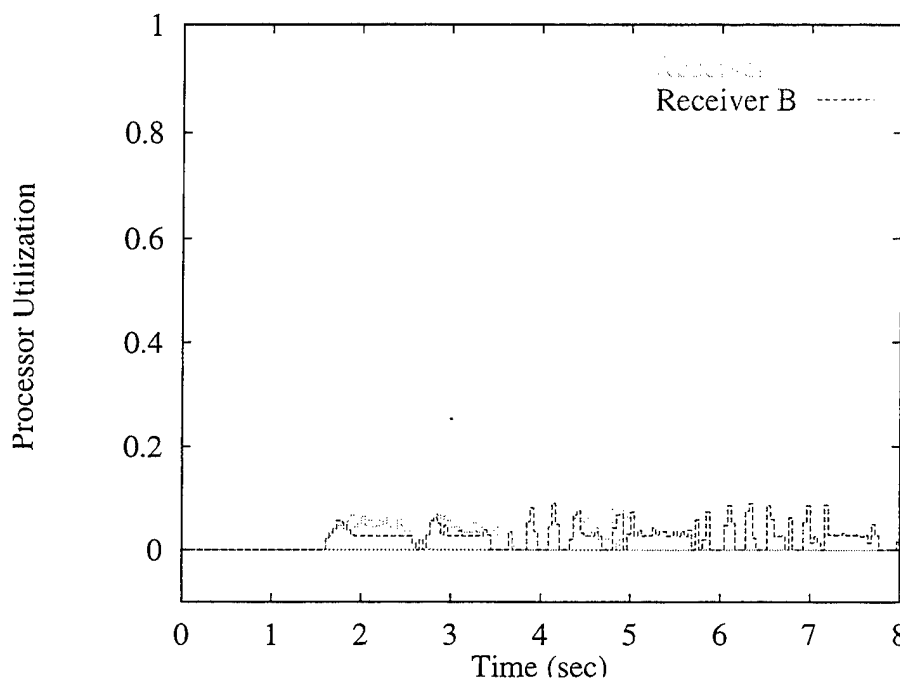


Figure 6-20: Experiment 17 Results

The results of Experiment 18 are shown in Figure 6-21. Again, a remote sender periodically sends packets to two receivers which have reservations and which use the `libsockets` implementation of sockets. The graph has time in seconds on the x-axis and processor utilization on the y-axis. The usage functions of the two receivers are plotted over the duration of the experiment. The average utilization for Receiver A in this case is 0.149 with a 5-percentile of 0.139 and a 95-percentile of 0.173. This indicates a fairly tight distribution. Receiver B has an average utilization of 0.130 with a 5-percentile of 0.115 and a 95-percentile of 0.140. Again the utilization achieved is quite consistent across periods for the duration of the test. It is clear that the receivers do not drop to very low utilizations for significant intervals of time. Their relatively constant usage indicates that they are able to process the incoming packets in a predictable manner.

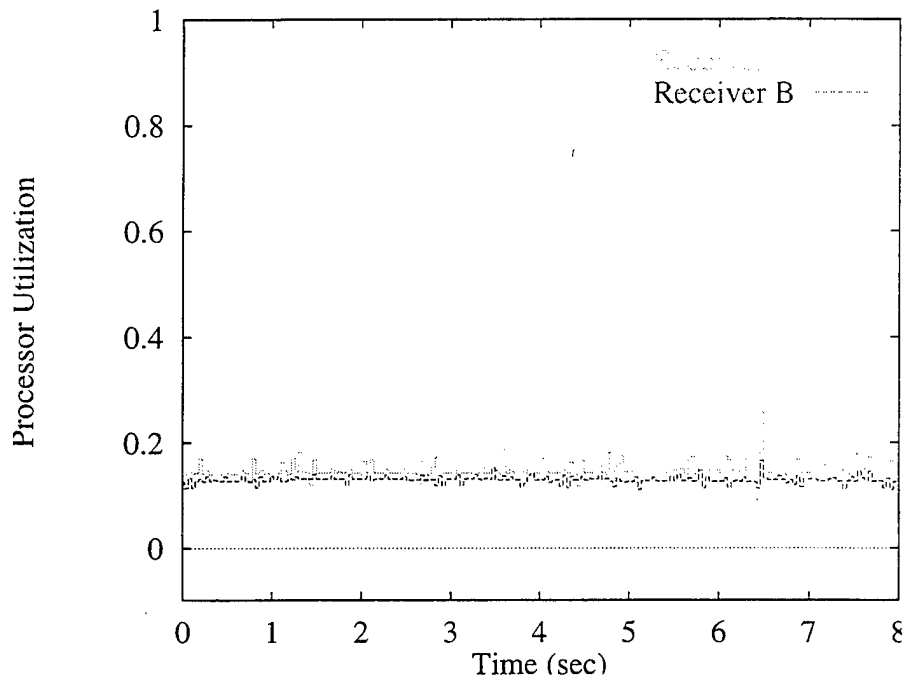


Figure 6-21: Experiment 18 Results

6.2.5.3 Analysis

The results from the `libsockets` experiments showed that when packet senders and receivers ran in time-sharing mode using the socket implementation provided by the UX Server, their behavior was erratic when other programs were competing for the processor and access to other services provided by the UX Server. Reserved packet senders and receivers that used `libsockets` for handling network packets had much better behavior. They were able to execute periodically and perform each sending or receiving computation by the end of the corresponding period. Other experiments (not presented here) showed that the behavior of reserved programs that used UX sockets was just as bad as that of unreserved programs with UX sockets. Also, unreserved programs that used `libsockets` exhibited unpredictable behavior when executing with competition as well. These experiments indicate that using a reservation mechanism or a `libsockets` mechanism alone does not ensure predictability in programs; both mechanisms are needed.

6.3 Scheduling cost

This section addresses the scheduling costs of predictable programs that can meet their timing constraints under the reservation system. It looks at measured aggregate costs for the system as well as measurements of scheduling operations that contribute to the costs. Such measurements enable cost projections to be made for specific task sets.

6.3.1 Measured aggregate scheduling cost

The reservation system ensures that reserved resources will be available to enable real-time programs to meet their timing constraints, but this predictability has costs associated with it. In particular, the accurate measurements and the timers necessary for enforcement take some time. In measuring the aggregate scheduling cost, the intention is to determine how much time is consumed by scheduling costs in the case of a reserved periodic thread compared to that of an unreserved thread.

6.3.1.1 Methodology

Experiment 19 is designed to measure the scheduling cost for a periodic thread as the period varies. The task set includes the periodic thread and an “idle” thread that runs in the background to consume all processor time not consumed by the periodic thread and the system’s housekeeping activities. The system scheduling cost is taken to be the total time of the test minus the time consumed by the periodic thread and the idle thread. This scheduling cost includes context switch times, associated cache effects, as well as the cost of timers and clock operations for the reservation mechanism.

One series of tests measures the scheduling cost for a reserved periodic thread whose reservation period ranges from 20 ms to 200 ms. Most of the cost for the reservation system is a fixed cost in each period, so a longer period implies a relatively smaller cost. The other series of tests measures scheduling cost for an unreserved periodic thread with a period that varies from 20 ms to 200 ms.

6.3.1.2 Results

The results of the scheduling cost measurements are presented in Figure 6-22. The graph consists of two functions: the scheduling cost associated with a reserved periodic thread as a function of period and the scheduling cost associated with an unreserved periodic thread as a function of period. The x-axis is the period in milliseconds, and the y-axis is the percentage of the processor that is lost to scheduling costs.

The scheduling cost for the reserved thread starts out about 3% for a reservation period of 20 ms and drops off as the reservation period is increased. For a 100 ms reservation period, the scheduling cost is about 0.5% and for a 200 ms reservation period, it is about 0.2%. For the unreserved thread, the scheduling cost is smaller, starting at about 2.2% for a 20 ms period. The cost drops off to about 0.5% for a 100 ms period and then to about 0.1% for a 200 ms period.

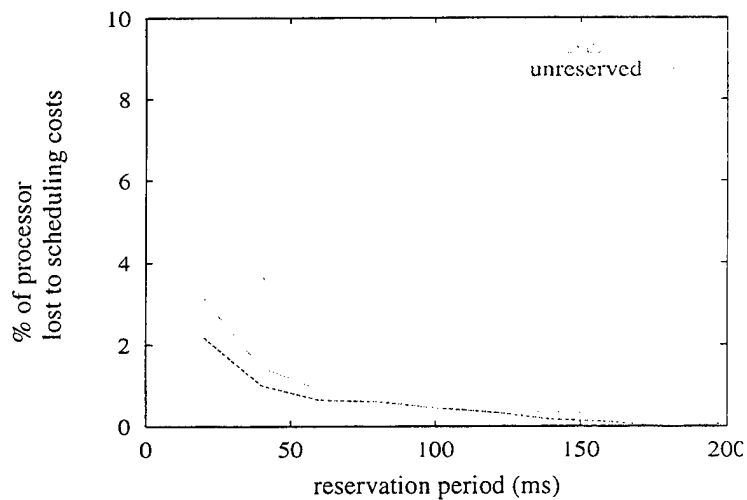


Figure 6-22: Scheduling Cost

6.3.1.3 Analysis

The scheduling cost measurements indicate that the cost of threads with very small reservation periods (smaller than about 30 ms) grows somewhat as the reservation period decreases. For reservation periods in the range of 40 to 100 ms, which would be an appropriate range for many audio and video applications for example, the scheduling cost is acceptable.

The scheduling cost is relatively high for the reservation system because the clock/timer card used in the experiments is very sensitive to the timing of loads and stores in its control and data registers. The card is used quite often in the reservation system to read a free-running clock, set an interrupting timer, or cancel a timer. Since each of these operations requires multiple reads and stores to device registers and since the device driver for the card contains many delay loops required to synchronize properly with the card, much time is wasted. With clock and timer support from a better card, the scheduling cost should be significantly lower.

6.3.2 Individual operations

This section presents measurements of the individual internal operations used by the reservation mechanism. These measurements can be used to project scheduling costs for task sets.

6.3.2.1 Reserve Switch

During a context switch, the system must switch the reserve to which it is charging computation time as it switches the thread that is running on the processor. This involves updat-

ing usage accumulators in the old reserve and possibly setting the overrun timer for the next reserve. The four measured cases for the reserve switch are:

- Neither the old or new activity was reserved - Just update the usage accumulators.
- The old activity was reserved - Cancel the overrun timer for the old activity and update usage.
- The new activity is reserved - Set up and arm the overrun timer for the new activity and update usage
- Both activities are reserved - Cancel the old overrun timer, set up the new overrun timer, and update usage.

The following table gives the measurements for these cases.

Action	Duration
Neither old nor new activity reserved	23 μ s
Old activity reserved	100 μ s
New activity reserved	100 μ s
Both activities reserved	180 μ s

Table 6-18: Reserve Switch

6.3.2.2 Overrun and Replenishment Timers

The measured cost for handling an overrun timer includes identifying the timer, setting some state in the reserve, and initiating a context switch. It does not include the cost of the context switch itself.

The cost for handling a replenishment timer includes identifying the timer, setting some state in the reserve, resetting the timer for the next reservation period boundary, and possibly resetting the overrun timer. Four different cases for replenishment timer handling were measured:

- A reserve with a reservation whose computation allocation had been depleted during the period (the overrun timer had expired for this activity), and it was waiting for a new allocation.
- A reserve with a reservation whose computation allocation had not been depleted.
- A reserve with no reservation whose replenishment timer expired while it was running.
- A reserve with no reservation which was not running at the time the replenishment timer expired.

The following table shows the measurements for the overrun timer and each of the replenishment timer cases.

Action	Duration
Handle overrun timer	130 μ s
Reserved and waiting for new allocation	170 μ s
Reserved but not waiting	140 μ s
Unreserved and running when timer expired	140 μ s
Unreserved and not running when timer expired	140 μ s

Table 6-19: Replenishment Timer

6.3.2.3 Usage checkpoints

This section gives measurements of the system primitives that extract usage information from the kernel. The reserve usage data from the reservation system implemented in RT-Mach come in two forms: the current accumulated usage of a reserve, and the accumulated usage as of the last reservation period boundary. The record of the accumulated usage of a reserve taken at a reservation period boundary is called a checkpoint. The reservation system offers two system primitives for retrieving usage data: the first gives the data for a single checkpoint along with the current accumulated usage, and the second give the last 20 checkpoints from the last 20 reservation period boundaries. The following table gives the measured costs for both of these system primitives.

Action	Duration
Retrieve single checkpoint	130 μ s
Retrieve 20 recent checkpoints	220 μ s

Table 6-20: Checkpoint Cost

6.3.2.4 Analysis

The measurements described in this section can be used to estimate the cost associated with running specific task sets on the reservation system. The replenishment timer costs occur in every period of every reserved activity, and these numbers can help project the impact of having many tasks with small reservation period. The overrun timer costs detail the penalty associated with a program whose computation does not closely match its reservation. The checkpoint costs can be used in estimating the overhead for a monitor and in choosing timing properties of a monitor to balance accuracy of information with overhead cost.

6.4 Chapter summary

This chapter addressed the questions of whether the reservation system supports predictable application behavior and how much the predictability costs in terms of scheduling overhead. To show that the reservation system supports predictable behavior for applications, several experiments were done using task sets consisting of independent compute-bound synthetic workloads as well as client/server task sets with synthetic workloads. In an experiment with three reserved programs and five unreserved competitors, the reserved programs achieved measured processor utilizations that had 5-percentiles and 95-percentiles within 3-7% of their average utilizations, indicating that they were able to get their processor reservations with very little variance in their computation times. In the client/server experiments, even a case where an unreserved client was competing with two reserved clients for the same server (along with other independent unreserved programs competing for the processor), the reserved clients were able to achieve their timing constraints. The 5-percentiles and 95-percentiles for the two reserved clients in this case were within 5% of their average utilizations. These experiments showed that the reservation system guarantees very tight distributions of processor utilization even with different types of computation and with different combinations of client/server interactions.

Additional experiments used a reserved QuickTime video player and a modified version of the X Server to show that reserved applications can coordinate with shared servers to satisfy timing constraints on computations such as displaying video frames. In these experiments, the reserved players (even with interference from competing non-real-time X clients) had processor utilization measurements with 5-percentiles and 95-percentiles within 65% of their average utilizations. The utilization distributions were not as tight as the synthetic client/servers because the internal structure of the X Server is not ideally suited to reserve propagation and charging to clients' reserves. However, the performance of the reserved players was still much improved over that of unreserved players which had measurements with 5-percentiles and 95-percentiles that were as much as 166% of their average utilizations.

Experiments with libsockets showed that with an appropriate protocol processing structure, packet senders and receivers could achieve predictable behavior. The reserved senders in the experiments had processor utilization measurements with 5-percentiles and 95-percentiles that were within 7% of the average utilizations, and the reserved receivers had 5- and 95-percentiles that were within 16% of the average utilizations. This is compared with unreserved senders and receivers that had 5- and 95-percentiles of up to 360% of their average utilizations.

The scheduling costs of the reservation system were measured by running a periodic thread and measuring the idle time left over to find the scheduling cost. The results from several experiments with reserved and unreserved periodic threads with different periods show that the scheduling cost of reserved threads is typically about twice that of unreserved threads.

Chapter 7

Related Work

The work presented in this dissertation draws on research in several different areas. The reserve model depends on theoretical results from real-time scheduling, and the design and implementation were influenced by the work in real-time system design as well as the requirements for multimedia applications. This chapter presents an overview of related work: most of the related work focuses on systems issues although a section on applications discusses work on tools and application adaptation techniques.

7.1 System Implementation

The increasing integration of computer and telecommunications technologies has focused attention on the real-time issues that arise in processing digital audio and video. Handling multimedia data streams requires an understanding of the timing requirements, encoding techniques, and data formats of the new media types [13,69,94,135,136] as well as programming interfaces [60,101] and new application design techniques [35].

7.1.1 Multimedia support

A great deal of recent work has focused on how software systems (including operating systems) can be designed to support multimedia applications. Many researchers and practitioners consider resource reservation desirable if not absolutely necessary for real-time multimedia operating systems [3,46,53,60,87,102,127]. Herrtwich [42] gives an argument for resource reservation and careful scheduling in these systems. Others prefer a best effort approach to OS design for multimedia applications [19,21,90]. Recent survey papers [116,132] and books [12,115] discuss work in this area.

One of the simplest ways to do resource allocation for multimedia applications is to dedicate an entire machine to a single multimedia application. Multimedia applications for single-user personal computers typically assume that only a single multimedia activity will exist at any one time. Or if several multimedia activities exist, the assumption is that they

will be associated with a single application that can do cooperative scheduling of these activities. Adobe Premiere [95] is an example of such an application. If this assumption is violated, neither the applications nor the system will be in a position to assert anything about the behavior of multimedia applications. There is no admission control or any kind of overload protection in such systems.

Other systems such as OS/2 [60] put limits on the number of high-level activities, such as video streams being processed by the system, as a primitive form of admission control. This technique does not address the problem of resource allocation or admission control for arbitrary computations and media processing applications.

Jeffay *et al.* implemented an operating system designed for guaranteed real-time scheduling [53]. A video capture and playback application demonstrates that the analytical techniques can be successfully applied to real applications. In subsequent work, Jeffay has focused attention on transport mechanisms to detect and deal with variability in network behavior [54]. The reservation system described in this dissertation uses real-time scheduling analysis as does Jeffay. It focuses on enforcement whereas Jeffay's recent work focuses more on flexibility. The reservation system would benefit greatly from the increased flexibility of having adaptive mechanisms such as Jeffay's at higher levels.

Anderson *et al.* [3] argue for introducing more sophisticated timing and scheduling features into operating systems, and their DASH system design supports a reservation model based on linear bounded arrival processes (LBAP) [22,23]. They implemented their system design and were able to report some preliminary experiences with the system. They use earliest deadline scheduling for real-time traffic because it is optimal in the sense that if any algorithm can schedule a particular collection of tasks, the earliest deadline algorithm can do it. Admission control for this system is based on a time-line where new jobs are admitted only if they fit onto the time-line when the job request arrives [129]. The reservation system described in this dissertation uses an admission control algorithm based on a periodic scheduling framework with scheduling analysis rather than a timeline approach. One way for reserves to accommodate one-shot events would be to use a timeline based admission control policy and scheduling algorithm. The reservation system focuses more on enforcement of specified computation times.

Hyden [46] considered the problem of supporting QOS in operating systems in his thesis. He implemented a system that offers a virtual processor interface to applications. A video decoder application demonstrates how such an interface can be used by an application. In contrast, the reserve system provides flexibility in reservation binding for a more integrated view of resource usage reservation, measurement, and enforcement.

Coulson *et al.* [21] present a system design based on Chorus [103]. This system uses earliest deadline scheduling, but they do not provide any admission control and usage enforcement. QOS commitments can be revoked, and overload is permitted; commitments are degraded as a response to overload. The work focuses primarily on fast context switching and reducing protection domain crossings. The reserve system provides guaranteed resource reservation using an enforcement mechanism with QOS policy modules layered over the reservation abstraction.

Jones [56] describes some ideas on system design for multimedia applications which depend on value functions as a means of scheduling processes based on timing constraints, semantic importance, user preferences, and other information about application-level requirements. The reserve system divides the QOS provision problem into a reservation mechanism and QOS policy layers. Jones' work focuses more on the QOS policy and could benefit from mechanisms for guaranteed resource reservation.

Many multimedia cards and co-processor boards and boxes are actually embedded systems with their own processors and operating systems. These systems must manage resources for multiple real-time activities that must be multiplexed.

Hopper [44] described Pandora's Box which is a transputer system designed to be a multimedia peripheral for a traditional workstation. The Pandora system employs several transputers, each of which handles a particular function in the system such as compression, decompression, network traffic, and audio. A similar approach is being pursued in the context of the Desk Area Network [40] and related operating system efforts [9,81]. The reserve system addresses the sharing of devices and software resources by appropriate admission control policies, scheduling algorithms, and enforcement mechanisms. Multiplexing of resources results in more efficient resource utilization.

The Mwave system [47] consists of a digital signal processor (DSP) card intended for use with a PC. The Mwave card handles various audio processing and telephony tasks, depending on the host processor for control functions. A programmer can develop applications that are divided between the host system and the Mwave processor. There is a programming environment that supports application development, and the operating system running on the Mwave processor provides some real-time support for scheduling and enforcement. In particular, the Mwave/OS performs enforcement functions, resetting the card when any of the real-time activities misses a deadline. The reserve abstraction provides more functionality for flexible binding of reserves, and the reserve system also has a more flexible policy for handling timing failures. In particular, real-time activities are not affected by the timing failure of an independent real-time or non-real-time activity.

7.1.2 QOS architectures

The reserve abstraction is designed to support higher-level architectures for managing quality of service specification and negotiation. Several QOS architectures have been proposed assuming that the mechanisms for operating system resource management, such as the reserve system, would be developed.

Nicolaou [89] described a QOS architecture suitable for programming distributed multimedia applications. His work describes an architecture that one might use to design and implement multimedia applications. While some systems issues like resource management and scheduling are identified as being important, the cited work does not address those problems. A prototype implementation based on the architecture demonstrates the feasibility of this approach, but since the prototype is implemented on UNIX, its performance suffers from a lack of real-time scheduling techniques in the operating system. The reserve

model is designed to provide the kind of operating system support necessary for Nicolaou's QOS architecture.

Wolf *et al.* [137] defined a QOS architecture for a communication transport system, and they have an initial implementation of their system, called the Heidelberg Transport System. The reserve system focuses more on admission control, dealing with interaction between processes (such as client/server interactions) and usage enforcement.

The QOS Broker [84] provides an architecture for handling QOS negotiation among resource "buyers" and "sellers." Protocols are provided for carrying out the negotiation, and a version of the QOS Broker was implemented and used in the context of a telerobotics application. The Broker contains hooks for reserving resources in operating systems and networks when reservation mechanisms are available. The reserve system described in this dissertation provides the mechanism necessary for negotiating guaranteed service.

7.1.3 Networking

The idea of bandwidth reservation for network quality of service models has been explored by a number of researchers. The thesis work complements the work on networking by providing resource reservation in the end hosts as well as in the network. Thus the operating system resource reservation work enables predictable end-to-end performance for real-time programs.

Ferrari and Verma [31] describe a model for guaranteeing bandwidth in a wide-area network. Their analytical model provided a basis for subsequent work on network bandwidth reservation. In contrast, Clark *et al.* [18] describe another service model based on the idea of predictive service. In related work, L. Zhang *et al.* [140] gave a basic description of RSVP, a protocol for reserving resources across nodes in an internetwork. This thesis provides operating system support to implement these types of schemes. Reserves support guaranteed service for hard real-time applications as well as supporting predictive service for soft real-time applications that dynamically change their requirements on various system resources.

H. Zhang [141] describes a bandwidth reservation model and an implementation in an internetwork protocol. He was able to allocate and control network bandwidth in gateways, but did not address the allocation and control of resources like the processor in the more general operating system environment. The reserve system would make it possible to address resource reservation issues in general end systems, extending H. Zhang's work on resource reservation within the network and its routers.

Anderson *et al.* [4] describe a Session Reservation Protocol (SRP) which reserves resources along the route of a connection to ensure a particular bandwidth and delay for the connection. This protocol provides resource reservation at routers for predictable network performance. The reserve system would make it possible to reserve resources at end hosts as well as in the network routers.

7.2 Scheduling theory

Real-time scheduling theory has been an active area of research for years. Many important theoretical results date back to the 1950's and 1960's [20,49], and work on system design dates back to the same time period [71,73,82]. More recently, real-time systems research has focused on scheduling algorithms and techniques for building complex, distributed real-time systems [1,130,131].

The reserve system was designed using several results from real-time scheduling theory in its admission control policy, scheduling algorithm, and synchronization and communication primitives. The reserve model is based on the original rate monotonic scheduling analysis [67] as well as recent extensions [63]. The simple two-parameter reservation model is suitable for a wide range of applications, but applications that require different reservation semantics might need a model that takes advantage of other scheduling algorithms and analyses. For example, an application might require generalized rate monotonic analysis [39,64,107,109], aperiodic servers with different replenishment policies [111,112,120], earliest deadline first scheduling [27,49,67], sporadic task scheduling [51], or deadline monotonic scheduling [7].

The reservation system also depends of priority inheritance protocols for fixed priority scheduling [96,108], and similar inheritance protocols for earliest deadline first [8,16] would be required for a reservation model based on that policy.

The two-parameter periodic scheduling framework of the original rate monotonic scheduling analysis divides the capacity of the processor among multiple tasks. Other scheduling techniques such as processor sharing and fair share scheduling aim to provide a similar kind of proportional sharing of the processor. The primary difference is that the two-parameter periodic scheduling framework specifies a granularity of sharing by specifying a period. Processor sharing and fair share scheduling seek to support sharing at a very fine granularity.

A processor sharing technique [50] intended to be accurate enough to accommodate the timing constraints of arbitrary multimedia applications would require a very small quantum. This would imply a high scheduling overhead. Such a system would also require some method for controlling the effects of synchronization and communication between processes.

Fair share schedulers [41,58,138] provide for resource allocation like processor sharing, although at a coarser granularity. Such schedulers ensure that users who pay more for compute time get better service than others who pay less do. They record usage measurements to try to match usage with target allocation levels over the long term.

More recently, work on fair share scheduling and proportional share scheduling has addressed the integration of network scheduling and end-system scheduling [118,133,134]. Instead of focusing on ensuring that a certain amount of computation time will be available by a deadline, this work focuses on ensuring that a certain proportion of the processor is available to a compute-bound task in any interval.

7.3 Applications

Two classes of applications turn out to be very important to the resource reservation work. Adaptive applications are important because dynamic real-time applications must be very sensitive to the relationship between their (changing) resource requirements and their levels of resource reservations. In addition, design tools, performance monitors, and dynamic resource allocation tools are necessary for the design and on-line monitoring and tuning of resource reservations for dynamic real-time applications.

7.3.1 Adaptive applications

Recently there has been a focus on how systems and applications can adapt their computations to the resources they find available to them. For example, video compression algorithms are designed to allow for tradeoffs in resource requirements in various ways. Software techniques are also being developed, primarily in the area of mobile computing, for application awareness of resource requirements and adaptation based on system resources available.

The MPEG compression algorithms support several ways to trade off between bandwidth, computational resources, and image quality [17,61]. The MPEG-1 q-factor trades image quality for less bandwidth and computational resources, and the MPEG-2 hierarchical encoding scheme supports incremental improvements in image quality for additional bandwidth and computation time [10,17]. Other methods such as subband encoding [121], Hyden's method [46], and other hierarchical encoding schemes [2,17] support this tradeoff as well.

Recent work in mobile computing explores how applications can be sensitive to the resources that are available to them in terms of network bandwidth, processor power, and screen resolution among others [32,91,106]. These applications discover the resources that are available to them and then use this information to guide their own computations. For example, a video player residing on a high-end workstation might request from a video server a full color (24 bits per pixel), full motion (30 frames per second), relatively high resolution (640x480 pixels) video stream. The same video player on a low-end personal computer might request only 256 bits of color, 15 frames per second, and a resolution of 160x120 pixels. Supporting a scenario like this requires that all of the involved system components are aware of the resources they have available, the resources they need to do their work, and the level at which all of the other components in the end-to-end activity can perform.

7.3.2 Tools

The reserve system provides a mechanism for tools that monitor and control resource usage. This section discusses current tools for monitoring functions and for controlling resource usage.

Different tools intended for monitoring various aspects of system performance work at different levels. Some are intended for program design while others are intended for system-level debugging and on-line resource monitoring.

Performance analysis tools such as gprof [29,37] and PCA [28] use PC sampling techniques to characterize program runtime behavior and object code to determine the static structure. This method gives a great deal of insight into the behavior of individual programs, but there is no notion of tuning task sets as a whole. Also, the method of exercising control of the programs being analyzed is through the programs themselves, either changing their structure or modifying their parameters. The tool does not exercise this control directly.

System monitoring tools typically separate the functions of capturing performance data, analyzing the data, and having an effect on the system that was measured. The Advanced Real-Time Monitor (ARM) [123], which was originally designed for the ARTS Kernel [124] and more recently updated for RT-Mach [125], takes this approach. ARM uses a kernel mechanism to capture scheduling events and then sends those events over the network to an ARM application running on a different machine. ARM then displays a scheduling history based on the events, and this history can be viewed, analyzed, and saved for future use.

Tools like xload [74] provide a very simple view of the cumulative processor load on a workstation. The xload application does this on-line, but it leaves out some interesting information like a breakdown of which processes are consuming what percent of the load. It also lacks a control element to help the user have an effect on the load through the tool.

Tools like the Memory Sizer on the Macintosh [6] offer control over a system resource, but the resource information is very simple, and one cannot set the memory size of a program while it is running. The reserve system allows for much more sophisticated control of system resource allocation. With the help of a QOS manager, a tool such as rmon can graphically display resource usage information and interact with the console user to control resource allocation.

Chapter 8

Conclusion

This dissertation has presented a comprehensive model describing resource reserves, the operations they support, the scheduling algorithm and enforcement mechanism required, and how reservations for various resource types can be encapsulated in a single framework. An implementation and experimental evaluation demonstrate that real applications with non-trivial client/server interactions can achieve predictable real-time performance using resource reserves. The reserves ensure this predictability even when there are multiple real-time and non-real-time applications competing for the same resources.

This work shows that system mechanisms that address entire activities are important for real-time resource management. Furthermore, it shows that enforcement is essential, otherwise a reservation abstraction has no meaning. Programming techniques such as software pipeline architectures with synchronized periods and deadlines are useful for achieving predictable behavior. The following sections detail the contributions of this work and directions that this work opens for future research.

8.1 Contributions

The contributions of this work include the abstraction for operating system resource reservation, its implementation, real applications which use it, and an experimental evaluation of those applications. The following sections discuss these in more detail.

8.1.1 Resource reservation abstraction

The resource reserve abstraction provides a model for how real-time scheduling algorithms and analyses can be incorporated in an operating system design in an integrated manner. This is in contrast to the specification of scheduling algorithms and analyses in the context of a simplified task model where many practical systems issues and programming issues are ignored. Two key features of the abstraction are flexible binding of reserves to threads and enforcement of reservation parameters.

Since reserves are first class objects in the system rather than being tightly and permanently bound to threads (or processes), the management of resources is much easier. For example, reservation parameters can be allocated for a reserve by a thread and then a reference to the reserve can be passed to system service providers invoked by the thread. By allowing the binding of reserves to threads to be flexible, reserves can be passed around in this way, and the resource usage for the abstract activity is tracked and guaranteed throughout all of the server calls.

The enforcement mechanism eases program development and debugging for programmers of hard and soft real-time applications. Programmers of hard real-time applications can exploit the usage accumulation mechanism to measure the requirements of their code during development. During runtime, the enforcement mechanism and usage measurements can be used to isolate timing bugs. Programmers of soft real-time applications can use the enforcement mechanism to ensure isolation between applications and to provide information on resource usage requirements for adaptive applications. This relieves the programmer of doing the exhaustive measurements and analysis usually required to achieve real-time predictability.

8.1.2 Implementation

The implementation of processor reserves in Real-Time Mach and the implementation of several real applications that use reserves demonstrate the feasibility of the approach described in this document. The implementation shows how to design an enforcement mechanism and integrate it with the scheduling policy. It shows how a reserve propagation mechanism can be built to ensure consistent resource reservation and account for abstract activities that span multiple threads (or processes). It also demonstrates how QOS managers can be incorporated into the system to negotiate reservation parameters between reserved applications and the operating system, and how resource usage monitor and control can be used to promote awareness of resource requirements and dynamic adjustment of resource allocation.

8.1.3 Experimental evaluation

The experimental evaluation demonstrates that the applications using the reservation system can achieve predictable behavior with acceptable overhead costs. Experiments with synthetic benchmark applications were able to achieve very consistent real-time performance even in the face of competition from other real-time and non-real-time applications. In the experiments, periodic reserved applications ran over a relatively long duration of time, and measurements of the processor utilization during each period were recorded. The 5-percentile and 95-percentile numbers for these measurements were typically within 5-7% of the average utilization across all the periods, indicating a very tight distribution of processor utilization measurements across periods and a quite consistent pattern of real-time behavior. In other experiments with real applications such as a video player and X Server, processor utilization measurements yielded 5- and 95-percentiles which differed from the average utilization by up to 65%. This is not nearly as tight as the synthetic client/server benchmark applications due to the input/output organization of the X Server which is not

ideally suited to reserve propagation techniques. In any case, the behavior of reserved X clients was much better than that of unreserved X clients, which had 5- and 95-percentiles that were as much as 166% of their average utilizations. Experiments with network transmit/receive applications showed processor utilization measurements with 5- and 95-percentiles of 7-16% of the average utilizations. When unreserved, these applications had measurements with 5- and 95-percentiles of up to 360% of their average utilizations. In each case, the reserved application showed much more consistent real-time behavior than its unreserved counterpart.

8.2 Future directions

The work described in this document opens up many avenues for future research. By providing a general framework and testbed platform for operating system resource reservation, this work provides a concrete context for many research topics such as QOS provision and negotiation, resource allocation algorithms, and adaptive application programming techniques.

This work on resource reservation also provides a design point for comparison with other system design approaches [65] and for other scheduling paradigms [119]. The idea of enforced resource reservation can also be used as a building block for exploring higher-level concerns such as the role of user in resource management [57].

Bibliography

- [1] A. K. Agrawala, K. D. Gordon, and P. Hwang, editors. *Mission Critical Operating Systems*. IOS Press, Amsterdam, 1992.
- [2] D. Anastassiou. Digital Television. *Proceedings of the IEEE*, 82(4):510–519, April 1994.
- [3] D. P. Anderson, S. Tzou, R. Wahbe, R. Govindan, and M. Andrews. Support for Continuous Media in the DASH System. In *Proceedings of the 10th International Conference on Distributed Computing Systems*, pages 54–61, May 1990.
- [4] D. P. Anderson, R. G. Herrtwich, and C. Schaefer. SRP: A Resource Reservation Protocol for Guaranteed-Performance Communication in the Internet. Technical Report TR-90-006, International Computer Science Institute, February 1990.
- [5] D. P. Anderson and R. Kuivila. A System for Computer Music Performance. *ACM Transactions on Computer Systems*, 8(1), February 1990.
- [6] Apple Computer, Inc. *Macintosh User's Guide*, 1991.
- [7] N. C. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings. Hard Real-Time Scheduling: The Deadline-Monotonic Approach. In *Eighth IEEE Workshop on Real-Time Operating Systems and Software*, pages 133–137, May 1991.
- [8] T. P. Baker. Stack-based Scheduling of Real Time Processes. *The Journal of Real-Time Systems*, 3(1):67–99, March 1991.
- [9] P. Barham, M. Hayter, D. McAuley, and I. Pratt. Devices on the Desk Area Network. *IEEE Journal on Selected Areas in Communications*, 13(4):722–732, May 1995.
- [10] S. Baron and W. Robin Wilson. MPEG Overview. *SMPTE Journal*, 6(103):391–394, June 1994.
- [11] J. Boykin, D. Kirschen, A. Langerman, and S. LoVerso. *Programming under Mach*. Addison-Wesley, 1993.
- [12] J. F. K. Buford, editor. *Multimedia Systems*. ACM Press and Addison-Wesley, 1994.

- [13] J. Burger. *The Desktop Multimedia Bible*. Addison-Wesley, 1993.
- [14] A. Campbell, G. Coulson, and D. Hutchison. A Quality of Service Architecture. *Computer Communication Review*, 24(2):6–27, April 1994.
- [15] K. Chen. A Study on the Timeliness Property in Real-Time Systems. *The Journal of Real-Time Systems*, 3(3):247–273, September 1991.
- [16] M.-I. Chen and K.-J. Lin. A Priority Ceiling Protocol for Multiple-Instance Resources. In *Proceedings of the Twelfth IEEE Real-Time Systems Symposium*, pages 140–149, December 1991.
- [17] T. Chiang and D. Anastassiou. Hierarchical Coding of Digital Television. *IEEE Communications Magazine*, 5(32):38–45, May 1994.
- [18] D. D. Clark, S. Shenker, and L. Zhang. Supporting Real-Time Applications in an Integrated Services Packet Network: Architecture and Mechanism. In *Proceedings of the SIGCOMM '92 Symposium on Communications Architectures and Protocols*, pages 14–26, October 1992.
- [19] C. L. Compton and D. L. Tennenhouse. Collaborative Load Shedding for Media-Based Applications. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems (ICMCS)*, pages 496–501, May 1994.
- [20] R. W. Conway, W. L. Maxwell, and L. W. Miller. *Theory of Scheduling*. Addison-Wesley, 1967.
- [21] G. Coulson, G. S. Blair, and P. Robin. Micro-kernel Support for Continuous Media in Distributed Systems. *Computer Networks and ISDN Systems*, 26(10):1323–1341, July 1994.
- [22] R. L. Cruz. A Calculus for Network Delay, Part I: Network Elements in Isolation. *IEEE Transactions on Information Theory*, 37(1):114–131, January 1991.
- [23] R. L. Cruz. A Calculus for Network Delay, Part II: Network Analysis. *IEEE Transactions on Information Theory*, 37(1):132–141, January 1991.
- [24] H. Custer. *Inside Windows NT*. Microsoft Press, 1993.
- [25] R. B. Dannenberg. A Real Time Scheduler/Dispatcher. In *Proceedings of the International Computer Music Conference*, pages 239–242, 1988.
- [26] P. Dasgupta et al. The Design and Implementation of the Clouds Distributed Operating System. *Computing Systems*, 3(1):11–45, Winter 1990.
- [27] M. L. Dertouzos. Control Robotics: The Procedural Control of Physical Processes. In *Proceedings of the IFIP Congress*, pages 807–813, August 1974.
- [28] Digital Equipment Corporation. *VAX Performance and Coverage Analyzer User's Reference Manual*.
- [29] J. Fenlason and R. Stallman. *gprof: The GNU Profiler*. Free Software Foundation, Inc., 1988.

- [30] D. Ferrari. Client Requirements for Real-Time Communication Services. *IEEE Communications Magazine*, 28(11):65–72, November 1990.
- [31] D. Ferrari and D. C. Verma. A Scheme for Real-Time Channel Establishment in Wide-Area Networks. *IEEE Journal on Selected Areas in Communication*, 8(3):368–379, April 1990.
- [32] A. Fox, S. D. Gribble, E. A. Brewer, and E. Amir. Adapting to Network and Client Variability via On-Demand Dynamic Distillation. In *Proceedings of the Seventh ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*, October 1996.
- [33] G. Gallassi, G. Rigolio, and L. Verri. Resource Management and Dimensioning in ATM Networks. *IEEE Network Magazine*, 4(3), May 1990.
- [34] J. Gettys, P. L. Karlton, and S. McGregor. The X Window System, Version 11. *Software – Practice and Experience*, 20(S2):S2/35–S2/67, October 1990.
- [35] S. J. Gibbs and D. C. Tsichritzis. *Multimedia Programming: Objects, Environments and Frameworks*. ACM Press and Addison-Wesley, 1995.
- [36] D. Golub, R. W. Dean, A. Forin, and R. F. Rashid. Unix as an Application Program. In *Proceedings of Summer 1990 USENIX Conference*, June 1990.
- [37] S. L. Graham, P. B. Kessler, and M. K. McKusick. gprof: a Call Graph Execution Profiler. In *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*, pages 120–126, June 1982.
- [38] Rhan Ha and J. W. S. Liu. Validating Timing Constraints in Multiprocessor and Distributed Real-Time Systems. In *Proceedings of the 14th IEEE International Conference on Distributed Computing Systems*, June 1994.
- [39] M. G. Harbour, M. H. Klein, and J. P. Lehoczky. Timing Analysis for Fixed-Priority Scheduling of Hard Real-Time Systems. *IEEE Transactions on Software Engineering*, 20(1):13–28, January 1994.
- [40] M. Hayter and D. McAuley. The Desk Area Network. *ACM Operating Systems Review*, 25(4):14–21, October 1991.
- [41] G. J. Henry. The Fair Share Scheduler. *AT&T Bell Laboratories Technical Journal*, 63(8):1845–1858, October 1984.
- [42] R. G. Herrtwich. The Role of Performance, Scheduling, and Resource Reservation in Multimedia Systems. In A. Karshmer and J. Nehmer, editors, *Operating Systems of the 90s and Beyond*, number 563 in Lecture Notes in Computer Science, pages 279–284. Springer-Verlag, 1991.
- [43] P. Hood and V. Grover. Designing Real Time Systems in Ada. Technical Report 1123-1, SofTech, Inc., January 1986.
- [44] A. Hopper. Pandora - an experimental system for multimedia applications. *ACM Operating Systems Review*, 24(2):19–34, April 1990.

- [45] N. C. Hutchinson and L. L. Peterson. The x-Kernel: An Architecture for Implementing Network Protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, January 1991.
- [46] E. A. Hyden. *Operating System Support for Quality of Service*. PhD thesis, University of Cambridge, February 1994.
- [47] International Business Machines. *Mwave/OS User's Guide*, 1993.
- [48] Y. Ishikawa, H. Tokuda, and C. W. Mercer. Priority Inversion in Network Protocol Module. *Proceedings of 1989 National Conference of the Japan Society for Software Science and Technology*, October 1989.
- [49] J. R. Jackson. Scheduling a Production Line to Minimize Maximum Tardiness. Technical Report Research Report 43, Management Science Research Project, UCLA, 1955.
- [50] R. Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. John Wiley & Sons, 1991.
- [51] K. Jeffay. Analysis of a Synchronization and Scheduling Discipline for Real-Time Tasks with Preemption Constraints. In *Proceedings of the IEEE Real-Time Systems Symposium*, December 1989.
- [52] K. Jeffay, D. F. Stanat, and C. U. Martel. On Non-Preemptive Scheduling of Periodic and Sporadic Tasks. In *Proceedings of the IEEE Real-Time Systems Symposium*, December 1991.
- [53] K. Jeffay, D. L. Stone, and F. D. Smith. Kernel Support for Live Digital Audio and Video. *Computer Communications (UK)*, 15(6):388–395, July-August 1992.
- [54] K. Jeffay, D. L. Stone, and F. D. Smith. Transport and Display Mechanisms for Multimedia Conferencing Across Packet-Switched Networks. *Computer Networks and ISDN Systems*, 26(10):1281–1304, July 1994.
- [55] E. D. Jensen, C. D. Locke, and H. Tokuda. A Time-Driven Scheduling Model for Real-Time Operating Systems. In *Proceedings of the 6th IEEE Real-Time Systems Symposium*, December 1985.
- [56] M. B. Jones. Adaptive Real-Time Resource Management Supporting Composition of Independently Authored Time-Critical Services. In *Proceedings of the Fourth Workshop on Workstation Operating Systems (WWOS-IV)*, pages 135–139, October 1993.
- [57] M. B. Jones et al. Support for User-Centric Modular Real-Time Resource Management in the Rialto Operating System. In *Proceedings of the Sixth International Workshop on Network and Operating System Support for Digital Audio and Video*, November 1995.
- [58] J. Kay and P. Lauder. A Fair Share Scheduler. *CACM*, 31(1):44–55, January 1988.

- [59] T. Kitayama, T. Nakajima, and H. Tokuda. RT-IPC: An IPC extension for Real-Time Mach. In *Proceedings of the USENIX Symposium on Microkernels and Other Kernel Architectures*, pages 91–104, September 1993.
- [60] W. Lawton, B. Noe, and M. Lopez. *Developing Multimedia Applications Under OS/2*. John Wiley & Sons, 1995.
- [61] D. Le Gall. MPEG: A Video Compression Standard for Multimedia Applications. *CACM*, 34(4):46–58, April 1991.
- [62] S. J. Leffler, M. K. McKusick, M. J. Karels, and J. S. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System*. Addison-Wesley, 1989.
- [63] J. P. Lehoczky, L. Sha, and Y. Ding. The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior. In *Proceedings of the 10th IEEE Real-Time Systems Symposium*, pages 166–171, December 1989.
- [64] J. P. Lehoczky. Fixed Priority Scheduling of Periodic Task Sets with Arbitrary Deadlines. In *Proceedings of the 11th IEEE Real-Time Systems Symposium*, pages 201–209, December 1990.
- [65] I. Leslie et al. The Design and Implementation of an Operating System to Support Distributed Multimedia Applications. *IEEE Journal on Selected Areas in Communications*, September 1996.
- [66] J. Leung and J. Whitehead. On the Complexity of Fixed-Priority Scheduling of Periodic, Real-Time Tasks. *Performance Evaluation*, 2:237–250, 1982.
- [67] C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard Real Time Environment. *JACM*, 20(1):46–61, 1973.
- [68] C. D. Locke. Software Architecture for Hard Real-Time Applications: Cyclic Executives vs. Fixed Priority Executives. *The Journal of Real-Time Systems*, 4(1):37–53, March 1992.
- [69] A. Luther. *Digital Video in the PC Environment*. Intertext Publications and McGraw-Hill, 2nd edition, 1991.
- [70] C. Maeda and B. N. Bershad. Protocol Service Decomposition for High-Performance Networking. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 244–255, December 1993.
- [71] G. K. Manacher. Production and Stabilization of Real-Time Task Schedules. *JACM*, 14(3):439–465, July 1967.
- [72] C. Martel. Preemptive Scheduling with Release Times, Deadlines, and Due Times. *JACM*, 29(3), 1982.
- [73] J. Martin. *Programming Real-Time Computer Systems*. Prentice-Hall, 1965.
- [74] Massachusetts Institute of Technology. xload man page, 1988. X11 Window System, Release 4, Massachusetts Institute of Technology.

- [75] C. W. Mercer and H. Tokuda. An Evaluation of Priority Consistency in Protocol Architectures. In *Proceedings of the IEEE 16th Conference on Local Computer Networks*, pages 386–398, October 1991.
- [76] C. W. Mercer, S. Savage, and H. Tokuda. Processor Capacity Reserves for Multimedia Operating Systems. Technical Report CMU-CS-93-157, School of Computer Science, Carnegie Mellon University, May 1993.
- [77] C. W. Mercer, J. Zelenka, and R. Rajkumar. On Predictable Operating System Protocol Processing. Technical Report CMU-CS-94-165, School of Computer Science, Carnegie Mellon University, May 1994.
- [78] C. W. Mercer, S. Savage, and H. Tokuda. Processor Capacity Reserves: Operating System Support for Multimedia Applications. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems (ICMCS)*, pages 90–99, May 1994.
- [79] C. W. Mercer and R. Rajkumar. An Interactive Interface and RT-Mach Support for Monitoring and Controlling Resource Management. In *Proceedings of the IEEE Real-Time Technology and Applications Symposium*, May 1995.
- [80] D. Merusi. *Software Implementation Techniques: VMS, UNIX, OS/2, and MS-DOS*. Digital Press, 1992.
- [81] S. J. Mullender, I. M. Leslie, and D. McAuley. Operating-System Support for Distributed Multimedia. In *Proceedings of the Summer 1994 USENIX Conference*, pages 209–219, San Francisco, CA, June 1994.
- [82] R. R. Muntz and E. G. Coffman, Jr. Preemptive Scheduling of Real-Time Tasks on Multiprocessor Systems. *JACM*, 17(2):324–338, 1970.
- [83] K. Nahrstedt and R. Steinmetz. Resource Management in Networked Multimedia Systems. *IEEE Computer*, 28(5):52–63, May 1995.
- [84] K. Nahrstedt and J. M. Smith. The QOS Broker. *IEEE Multimedia*, 2(1):53–67, Spring 1995.
- [85] T. Nakajima and H. Tokuda. Implementation of Scheduling Policies in Real-Time Mach. In *Proceedings of the Second International Workshop on Object Orientation in Operating Systems*, pages 165–169, September 1992.
- [86] T. Nakajima, T. Kitayama, H. Arakawa, and H. Tokuda. Integrated Management of Priority Inversion in Real-Time Mach. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 120–130, December 1993.
- [87] T. Nakajima and H. Tezuka. A Continuous Media Application supporting Dynamic QOS Control on Real-Time Mach. In *Proceedings of the Second ACM International Conference on Multimedia*, pages 289–297, October 1994.
- [88] C. Nicolaou. An Architecture for Real-Time Multimedia Communication Systems. *IEEE Journal on Selected Areas in Communications*, 8(3), April 1990.

- [89] C. A. Nicolaou. *A Distributed Architecture for Multimedia Communication Systems*. PhD thesis, University of Cambridge, 1991. Available as University of Cambridge Computer Laboratory Technical Report No. 220.
- [90] J. Nieh and M. S. Lam. SMART: A Processor Scheduler for Multimedia Applications. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, page 233, December 1995.
- [91] B. D. Noble, M. Price, and M. Satyanarayanan. A Programming Interface for Application-Aware Adaptation in Mobile Computing. *Computing Systems*, 8(4). 1995.
- [92] S. Oikawa and H. Tokuda. Efficient Timing Management for User-Level Real-Time Threads. In *Proceedings of the IEEE Real-Time Technology and Applications Symposium*, pages 27–32, May 1995.
- [93] K. Patel, B. C. Smith, and L. A. Rowe. Performance of a Software MPEG Video Decoder. In *Proceedings of the First ACM International Conference on Multimedia*, pages 75–82, August 1993.
- [94] K. C. Pohlmann. *Principles of Digital Audio*. McGraw-Hill, 3rd edition, 1995.
- [95] *Adobe Premiere for Macintosh*, 1993.
- [96] R. Rajkumar. *Task Synchronization in Real-Time Systems*. PhD thesis, Carnegie Mellon University, August 1989.
- [97] R. Rajkumar. *Synchronization in Real-Time Systems: A Priority Inheritance Approach*. Kluwer Academic Publishers, 1991.
- [98] E. P. Rathgeb. Modeling and Performance Comparison of Policing Mechanisms for ATM Networks. *IEEE Journal on Selected Areas in Communications*, 9(3):325–334, April 1991.
- [99] J. F. Ready. VRTX: A Real-Time Operating System for Embedded Microprocessor Applications. *IEEE Micro*, 6(4):8–17, August 1986.
- [100] R. R. Riesz and E. T. Klemmer. Subjective Evaluation of Delay and Echo Suppressors in Telephone Communications. *The Bell System Technical Journal*, 42, 1963.
- [101] S. Rimmer. *Multimedia Programming for Windows*. Windcrest/McGraw-Hill, 1994.
- [102] T. Roscoe. *The Structure of a Multi-Service Operating System*. PhD thesis, University of Cambridge, April 1995.
- [103] M. Rozier et al. Overview of the CHORUS Distributed Operating System. In *Proceedings of the USENIX Workshop on Micro-kernels and Other Kernel Architectures*, pages 39–69, April 1992.
- [104] T. G. Saponas and R. B. Demuth. The Distributed iRMX Operating System: A Real-Time Distributed System. In A. K. Agrawala, K. D. Gordon, and P. Hwang, editors. *Mission Critical Operating Systems*, chapter 16, pages 208–231. IOS Press, Amsterdam, 1992.

- [105] J. E. Sasinowski and J. K. Strosnider. ARTIFACT: A Platform for Evaluating Real-Time Window System Designs. In *Proceedings of 16th IEEE Real-Time Systems Symposium*, pages 342–352, December 1995.
- [106] M. Satyanarayanan, B. Noble, P. Kumar, and M. Price. Application-aware Adaptation for Mobile Computing. *Operating Systems Review*, 29(1), January 1995.
- [107] L. Sha and J. B. Goodenough. Real-Time Scheduling Theory and Ada. *IEEE Computer*, 23(4):53–62, April 1990.
- [108] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, September 1990.
- [109] L. Sha, R. Rajkumar, and S. S. Sathaye. Generalized Rate-Monotonic Scheduling Theory: A Framework for Developing Real-Time Systems. *Proceedings of the IEEE*, 82(1):68–82, January 1994.
- [110] J. A. Smith. The Multi-Threaded X Server. *The X Resource*, 1:73–89, 1992.
- [111] B. Sprunt, L. Sha, and J. P. Lehoczky. Aperiodic Task Scheduling for Hard Real-Time Systems. *The Journal of Real-Time Systems*, 1(1):27–60, June 1989.
- [112] B. Sprunt. *Aperiodic Task Scheduling for Real-Time Systems*. PhD thesis, Carnegie Mellon University, 1990.
- [113] J. A. Stankovic and K. Ramamritham. The Design of the Spring Kernel. In *Proceedings of the IEEE Real-Time Systems Symposium*, December 1987.
- [114] J. A. Stankovic. Misconceptions About Real-Time Computing: A Serious Problem for Next-Generation Systems. *IEEE Computer*, 21(10), October 1988.
- [115] R. Steinmetz and K. Nahrstedt. *Multimedia: Computing, Communications, and Applications*. Prentice-Hall, 1995.
- [116] R. Steinmetz. Analyzing the Multimedia Operating System. *IEEE Multimedia*, 2(1):63–84, Spring 1995.
- [117] R. Steinmetz. Human Perception of Jitter and Media Synchronization. *IEEE Journal on Selected Areas in Communications*, 14(1), January 1996.
- [118] I. Stoica, H. Abdel-Wahab, and K. Jeffay. A Proportional Share Resource Allocation for Real-Time, Time-Shared Systems. Technical Report Technical Report 96-18, Old Dominion University, May 1996.
- [119] I. Stoica, H. Abdel-Wahab, and K. Jeffay. On the Duality between Resource Reservation and Proportional Share Resource Allocation. Technical Report Technical Report 96-19, Old Dominion University, May 1996.
- [120] J. K. Strosnider, J. P. Lehoczky, and L. Sha. The Deferrable Server Algorithm for Enhanced Aperiodic Responsiveness in Hard Real-Time Environments. *IEEE Transactions on Computers*, 44(1):73–91, January 1995.

- [121] D. S. Taubman. *Directionality and Scalability in Image and Video Compression*. PhD thesis, University of California at Berkeley, 1994.
- [122] K. W. Tindell, A. Burns, and A. J. Wellings. Mode Changes in Priority Pre-emptively Scheduled Systems. In *Proceedings of IEEE Real-Time Systems Symposium*, pages 100–109, December 1992.
- [123] H. Tokuda, M. Kotera, and C. W. Mercer. A Real-Time Monitor for a Distributed Real-Time Operating System. In *Proceedings of ACM SIGOPS and SIGPLAN workshop on parallel and distributed debugging*, May 1988.
- [124] H. Tokuda and C. W. Mercer. ARTS: A Distributed Real-Time Kernel. *ACM Operating Systems Review*, 23(3):29–53, July 1989.
- [125] H. Tokuda, T. Nakajima, and P. Rao. Real-Time Mach: Toward a Predictable Real-Time System. In *Proceedings of USENIX Mach Workshop*, pages 73–82, October 1990.
- [126] H. Tokuda, Y. Tobe, S. T.-C. Chou, and J. M. F. Moura. Continuous Media Communication with Dynamic QOS Control Using ARTS with an FDDI Network. In *Proceedings of the SIGCOMM '92 Symposium on Communications Architectures and Protocols*, pages 88–98. ACM, October 1992.
- [127] H. Tokuda and T. Kitayama. Dynamic QOS Control based on Real-Time Threads. In *Proceedings of the Fourth International Workshop on Network and Operating System Support for Digital Audio and Video*, pages 113–122, November 1993.
- [128] D. Towsley. Providing Quality of Service in Packet Switched Networks. In *Performance Evaluation of Computer and Communication Systems: Joint Tutorial Papers of Performance '93 and Sigmetrics '93*, pages 560–586, 1993.
- [129] S. Tzou, 1993. Personal Communication.
- [130] A. M. van Tilborg and G. M. Koob, editors. *Foundations of Real-Time Computing: Formal Specifications and Methods*. Kluwer Academic Publishers, 1991.
- [131] A. M. van Tilborg and G. M. Koob, editors. *Foundations of Real-Time Computing: Scheduling and Resource Management*. Kluwer Academic Publishers, 1991.
- [132] A. Vogel, B. Kerherve, G. von Bochmann, and J. Gecsei. Distributed Multimedia and QOS: A Survey. *IEEE Multimedia*, 2(2):10–19, Summer 1995.
- [133] C. A. Waldspurger and W. E. Weihl. Lottery Scheduling: Flexible Proportional-Share Resource Management. In *Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–11, November 1994.
- [134] C. A. Waldspurger and W. E. Weihl. Stride Scheduling: Deterministic Proportional-Share Resource Management. Technical Report Technical Memorandum MIT/LCS/TM-528, Massachusetts Institute of Technology Laboratory for Computer Science, June 1995.

- [135] J. Watkinson. *The Art of Digital Audio*. Focal Press, 1988.
- [136] J. Watkinson. *The Art of Digital Video*. Boston: Focal Press, 2nd edition, 1994.
- [137] L. C. Wolf and R. G. Herrtwich. The System Architecture of the Heidelberg Transport System. *ACM Operating Systems Review*, 28(2):51–64, April 1994.
- [138] C. M. Woodside. Controllability of Computer Performance Tradeoffs Obtained Using Controlled-Share Queue Schedulers. *IEEE Transaction on Software Engineering*, SE-12(10):1041–1048, October 1986.
- [139] M. Yuhara, B. N. Bershad, C Maeda, and J. E. B. Moss. Efficient Packet Demultiplexing for Multiple Endpoints and Large Messages. In *Proceedings of the 1994 Winter USENIX Conference*, January 1994.
- [140] L. Zhang, S. Deering, D. Estrin, S. Shenker, and D. Zappala. RSVP: A New Resource ReSerVation Protocol. *IEEE Network*, pages 8–18, September 1993.
- [141] H. Zhang. *Service Disciplines for Packet-Switching Integrated-Services Networks*. PhD thesis, University of California at Berkeley, 1993.

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Carnegie Mellon University does not discriminate and Carnegie Mellon University is required not to discriminate in admission, employment, or administration of its programs or activities on the basis of race, color, national origin, sex or handicap in violation of Title VI of the Civil Rights Act of 1964, Title IX of the Educational Amendments of 1972 and Section 504 of the Rehabilitation Act of 1973 or other federal, state, or local laws or executive orders.

In addition, Carnegie Mellon University does not discriminate in admission, employment or administration of its programs on the basis of religion, creed, ancestry, belief, age, veteran status, sexual orientation or in violation of federal, state, or local laws or executive orders. However, in the judgment of the Carnegie Mellon Human Relations Commission, the Department of Defense policy of, "Don't ask, don't tell, don't pursue," excludes openly gay, lesbian and bisexual students from receiving ROTC scholarships or serving in the military. Nevertheless, all ROTC classes at Carnegie Mellon University are available to all students.

Inquiries concerning application of these statements should be directed to the Provost, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213, telephone (412) 268-6684 or the Vice President for Enrollment, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213, telephone (412) 268-2056.

Obtain general information about Carnegie Mellon University by calling (412) 268-2000.